1 One-Time Pad & Kerckhoffs' Principle

You can't learn about cryptography without meeting Alice, Bob, and Eve. This chapter is about the classic problem of **private communication**, in which Alice has a message that she wants to convey to Bob, while also keeping the contents of the message hidden from an eavesdropper[1] Eve. You'll soon learn that there is more to cryptography than just private communication, but it is the logical place to start.

1.1 What Is [Not] Cryptography?

"To define is to limit."
—Oscar Wilde

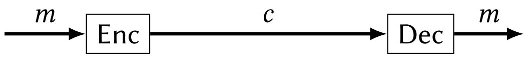
Cryptography is not a magic spell that solves all security problems. Cryptography can provide solutions to cleanly defined problems that often abstract away important but messy real-world concerns. Cryptography can give guarantees about what happens in the presence of certain well-defined classes of attacks. These guarantees may not apply if real-world attackers "don't follow the rules" of a cryptographic security model.

Always keep this in mind as we define (*i.e.*, limit) the problems that we solve in this course.

Encryption Basics & Terminology

Let's begin to formalize our scenario involving Alice, Bob, and Eve. Alice has a message m that she wants to send (privately) to Bob. We call m the **plaintext**. We assume she will somehow transform that plaintext into a value c (called the **ciphertext**) that she will actually send to Bob. The process of transforming m into c is called encryption, and we will use Enc to refer to the encryption algorithm. When Bob receives c, he runs a corresponding decryption algorithm Dec to recover the original plaintext m.

We assume that the ciphertext may be observed by the eavesdropper Eve, so the (informal) goal is for the ciphertext to be meaningful to Bob but meaningless to Eve.



[1] "Eavesdropper" refers to someone who secretly listens in on a conversation between others. The term originated as a reference to someone who literally hung from the eaves of a building in order to hear conversations happening inside.

Secrets & Kerckhoffs' Principle

Something important is missing from this picture. If we want Bob to be able to decrypt c, but Eve to *not* be able to decrypt c, then Bob must have some information that Eve doesn't have (do you see why?). Something has to be kept secret from Eve.

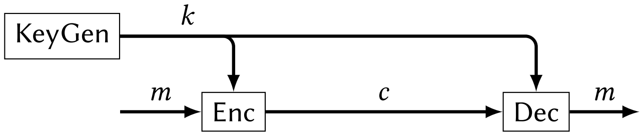
You might suggest to make the details of the Enc and Dec algorithms secret. This is how cryptography was done throughout most of the last 2000 years, but it has major drawbacks. If the attacker does eventually learn the details of Enc and Dec, then the only way to recover security is to *invent new algorithms*. If you have a system with many users, then the only way to prevent everyone from reading everyone else's messages is to *invent new algorithms* for each pair of users. Inventing even one good encryption method is already hard enough!

The first person to articulate this problem was Auguste Kerckhoffs. In 1883 he formulated a set of cryptographic design principles. Item #2 on his list is now known as **Kerckhoffs' principle**:

> **Kerckhoffs' Principle:**
>
> *"Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."*
>
> **Literal translation:** [The method] must not be required to be secret, and it must be able to fall into the enemy's hands without causing inconvenience.
>
> **Bottom line:** Design your system to be secure even if the attacker has complete knowledge of all its algorithms.

If the algorithms themselves are not secret, then there must be some other secret information in the system. That information is called the **(secret) key**. The key is just an extra piece of information given to both the Enc and Dec algorithms. Another way to interpret Kerckhoffs' principle is that *all of the security of the system should be concentrated in the secrecy of the key*, not the secrecy of the algorithms. If a secret key gets compromised, you only need to choose a new one, not reinvent an entirely new encryption algorithm. Multiple users can all safely use the same encryption algorithm but with independently chosen secret keys.

The process of choosing a secret key is called **key generation**, and we write KeyGen to refer to the (randomized) key generation algorithm. We call the collection of three algorithms (Enc, Dec, KeyGen) an **encryption scheme.** Remember that Kerckhoffs' principle says that we should assume that an attacker knows the details of the KeyGen algorithm. But also remember that knowing the details (i.e., source code) of a randomized algorithm doesn't mean you know the *specific output* it gave when the algorithm was executed.



Excuses, Excuses

Let's practice some humility. Here is just a partial list of issues that are clearly important for the problem of private communication, but which are not addressed by our definition of the problem.

- We are not trying to hide *the fact that Alice is sending something* to Bob, we only want to hide the *contents* of that message. Hiding the existence of a communication channel is called *steganography.*

- We won't consider the question of how c reliably gets from Alice to Bob. We'll just take this issue for granted.

- For now, we are assuming that Eve just passively observes the communication between Alice & Bob. We aren't considering an attacker that tampers with c (causing Bob to receive and decrypt a different value), although we will consider such attacks later in the book.

- We won't discuss *how* Alice and Bob actually obtain a common secret key in the real world. This problem (known as **key distribution**) is clearly incredibly important, and we will discuss some clever approaches to it much later in the book.

  In my defense, the problem we are solving is already rather non-trivial: once two users have established a shared secret key, how can they use that key to communicate privately?

- We won't discuss how Alice and Bob keep their key secret, even after they have established it. One of my favorite descriptions of cryptography is due to Lea Kissner (former principal security engineer at Google): *"cryptography is a tool for turning lots of different problems into key management problems."*

- Throughout this course we simply assume that the users have the ability to uniformly sample random strings. Indeed, without randomness there is no cryptography. In the real world, obtaining uniformly random bits from deterministic computers is extremely non-trivial. John von Neumann famously said, *"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."* Again, even when we take uniform randomness for granted, we still face the non-trivial question of how to *use* that randomness for private communication (and other applications), and also how to use only a *manageable amount* of randomness.

Not Cryptography

People use many techniques to try to hide information, but many are "non-cryptographic" since they don't follow Kerckhoffs' principle:

- Encoding/decoding methods like base64 ...

```
joy of cryptography   ↔   b25seSBuZXJkcyB3aWxsIHJlYWQgdGhpcw==
```

. . . are useful for incorporating arbitrary binary data into a structured file format that supports limited kinds of characters. But since base64 encoding/decoding involves no secret information, it adds nothing in terms of *security*.

- Sometimes the simplest way to describe an encryption scheme is with operations on binary strings (i.e., 0s and 1s) data. As we will see, one-time pad is defined in terms of plaintexts represented as strings of bits. (Future schemes will require inputs to be represented as a bitstring of a specific length, or as an element of $\mathbb{Z}_n$, etc.)

  In order to make sense of some algorithms in this course, it may be necessary to think about data being converted into binary representation. Just like with base64, representing things in binary has no effect on security since it does not involve any secret information. **Writing something in binary is not a security measure!**

1.2 Specifics of One-Time Pad

Good place to introduce abstractions type m type c etc.

People have been trying to send secret messages for roughly 2000 years, but there are really only 2 useful ideas from before 1900 that have any relevance to modern cryptography. The first idea is Kerckhoffs' principle, which you have already seen. The other idea is **one-time pad (OTP)**, which illustrates several important concepts, and can even still be found hiding deep inside many modern encryption schemes.

One-time pad is sometimes called "Vernam's cipher" after Gilbert Vernam, a telegraph engineer who patented the scheme in 1919. However, an earlier description of one-time pad was rather recently discovered in an 1882 text by Frank Miller on telegraph encryption.[2]

In most of this book, secret keys will be strings of bits. We generally use the variable λ to refer to the length (# of bits) of the secret key in a scheme, so that keys are elements of the set $\{0,1\}^\lambda$. In the case of one-time pad, the choice of λ doesn't affect security ($\lambda = 10$ is "just as secure" as $\lambda = 1000$); however, the length of the keys and plaintexts must be compatible. In future chapters, increasing λ has the effect of making the scheme harder to break. For that reason, λ is often called the **security parameter** of the scheme.

In one-time pad, not only are the keys λ-bit strings, but plaintexts and ciphertexts are too. You should consider this to be just a simple coincidence, because we will soon encounter schemes in which keys, plaintexts, and ciphertexts are strings of different sizes.

The specific KeyGen, Enc, and Dec algorithms for one-time pad are given below:

Construction 1.1 (One-time pad)

```
KeyGen:
    $k \leftarrow \{0,1\}^\lambda$
    return $k$

Enc($k, m \in \{0,1\}^\lambda$):
    return $k \oplus m$

Dec($k, c \in \{0,1\}^\lambda$):
    return $k \oplus c$
```

K <$ dist.

Recall that "$k \leftarrow \{0,1\}^\lambda$" means to sample k uniformly from the set of λ-bit strings. This uniform choice of key is the only randomness in all of the one-time pad algorithms. As we will see, all of its security stems from this choice of using the uniform distribution; keys that are chosen differently do not provide equivalent security.

↳ Uniform sampling from key space.

[2] See the article Steven M. Bellovin: "Frank Miller: Inventor of the One-Time Pad." *Cryptologia* 35 (3), 2011.

Example *Encrypting the following 20-bit plaintext m under the 20-bit key k using OTP results in the ciphertext c below:*

$$\begin{array}{rll} & \texttt{11101111101111100011} & (m) \\ \oplus & \texttt{00011001110000111101} & (k) \\ \hline & \texttt{11110110011111011110} & (c = \mathsf{Enc}(k, m)) \end{array}$$

Decrypting the following ciphertext c using the key k results in the plaintext m below:

$$\begin{array}{rll} & \texttt{00001001011110010000} & (c) \\ \oplus & \texttt{10010011101011100010} & (k) \\ \hline & \texttt{10011010110101110010} & (m = \mathsf{Dec}(k, c)) \end{array}$$

Note that Enc and Dec are essentially the same algorithm (return the xor of the two arguments). This results in some small level of convenience and symmetry when implementing one-time pad, but it is more of a coincidence than something truly fundamental about encryption (see Exercises 1.12 & 2.5). Later on you'll see encryption schemes whose encryption & decryption algorithms look very different.

Correctness

The first property of one-time pad that we should confirm is that the receiver does indeed recover the intended plaintext when decrypting the ciphertext. Without this property, the thought of using one-time pad for communication seems silly. Written mathematically:

Claim 1.2 *For all $k, m \in \{0,1\}^\lambda$, it is true that $\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$.*

Proof This follows by substituting the definitions of OTP Enc and Dec, then applying the properties of xor listed in Chapter 0.3. For all $k, m \in \{0,1\}^\lambda$, we have:

$$\begin{aligned} \mathsf{Dec}(k, \mathsf{Enc}(k, m)) &= \mathsf{Dec}(k, k \oplus m) \\ &= k \oplus (k \oplus m) \\ &= (k \oplus k) \oplus m \\ &= 0^\lambda \oplus m \\ &= m. \end{aligned}$$

Example *Encrypting the following plaintext m under the key k results in ciphertext c, as follows:*

$$\begin{array}{rll} & \texttt{00110100110110001111} & (m) \\ \oplus & \texttt{11101010011010001101} & (k) \\ \hline & \texttt{11011110101100000010} & (c) \end{array}$$

Decrypting c using the same key k results in the original m:

$$\begin{array}{rll} & \texttt{11011110101100000010} & (c) \\ \oplus & \texttt{11101010011010001101} & (k) \\ \hline & \texttt{00110100110110001111} & (m) \end{array}$$

Perfect secrecy game from lecture? Elaborate explanation with intro to games?

Security

Suppose Alice and Bob are using one-time pad but are concerned that an attacker sees their ciphertext. They can't presume what an attacker will do after seeing the ciphertext. But they would like to say something like, *"because of the specific way the ciphertext was generated, it doesn't reveal any information about the plaintext to the attacker, no matter what the attacker does with the ciphertext."*

We must first precisely specify how the ciphertext is generated. The Enc algorithm already describes the process, but it is written from the point of view of Alice and Bob. When talking about security, we have to think about what Alice and Bob do, but from the eavesdropper's point of view! From Eve's point of view, Alice uses a key that was chosen in a specific way (uniformly at random), she encrypts a plaintext with that key using OTP, and finally reveals only the resulting ciphertext (and not the key) to Eve.

More formally, from Eve's perspective, seeing a ciphertext corresponds to receiving an output from the following algorithm:

```
EAVESDROP($m \in \{0,1\}^\lambda$):
  $k \leftarrow \{0,1\}^\lambda$
  $c := k \oplus m$
  return $c$
```

.

It's crucial that you appreciate what this EAVESDROP algorithm represents. It is meant to describe **not what the attacker *does***, but rather the process (carried out by Alice and Bob!) that produces **what the attacker *sees***. We always treat the attacker as some (unspecified) process that receives output from this EAVESDROP algorithm. Our goal is to say something like "the output of EAVESDROP doesn't reveal the input m."

EAVESDROP is a *randomized* algorithm — remember that "$k \leftarrow \{0,1\}^\lambda$" means to sample k from the uniform distribution on λ-bit strings. If you call EAVESDROP several times, even on the same input, you are likely to get different outputs. Instead of thinking of "EAVESDROP(m)" as a single string, you should think of it as a *probability distribution* over strings. Each time you call EAVESDROP(m), you see a **sample** from that distribution.

Example *Let's take $\lambda = 3$ and work out by hand the distributions EAVESDROP(010) and EAVESDROP(111). In each case EAVESDROP chooses a value of k uniformly in $\{0,1\}^3$ — each of the possible values with probability 1/8. For each possible choice of k, we can compute what the output of EAVESDROP (c) will be:*

EAVESDROP(010):

Pr	*k*	*output c = k ⊕ 010*
1/8	000	010
1/8	001	011
1/8	010	000
1/8	011	001
1/8	100	110
1/8	101	111
1/8	110	100
1/8	111	101

EAVESDROP(111):

Pr	*k*	*output c = k ⊕ 111*
1/8	000	111
1/8	001	110
1/8	010	101
1/8	011	100
1/8	100	011
1/8	101	010
1/8	110	001
1/8	111	000

An introduction to game playing proofs [with formal games]

So the distribution $\textsc{eavesdrop}(\texttt{010})$ *assigns probabilty 1/8 to* $\texttt{010}$, *probability 1/8 to* $\texttt{011}$, *and so on.*

In this example, notice how every string in $\{\texttt{0},\texttt{1}\}^3$ appears *exactly once* in the c column of $\textsc{eavesdrop}(\texttt{010})$. This means that $\textsc{eavesdrop}$ assigns probability 1/8 to *every* string in $\{\texttt{0},\texttt{1}\}^3$, which is just another way of saying that the distribution is the *uniform distribution* on $\{\texttt{0},\texttt{1}\}^3$. The same can be said about the distribution $\textsc{eavesdrop}(\texttt{111})$, too. Both distributions are just the uniform distribution in disguise!

There is nothing special about $\texttt{010}$ or $\texttt{111}$ in these examples. For any λ and any $m \in \{\texttt{0},\texttt{1}\}^\lambda$, the distribution $\textsc{eavesdrop}(m)$ is the uniform distribution over $\{\texttt{0},\texttt{1}\}^\lambda$.

Claim 1.3 *For every* $m \in \{\texttt{0},\texttt{1}\}^\lambda$, *the distribution* $\textsc{eavesdrop}(m)$ *is the* ***uniform distribution*** *on* $\{\texttt{0},\texttt{1}\}^\lambda$. *Hence, for all* $m, m' \in \{\texttt{0},\texttt{1}\}^\lambda$, *the distributions* $\textsc{eavesdrop}(m)$ *and* $\textsc{eavesdrop}(m')$ *are identical.*

Proof Arbitrarily fix $m, c \in \{\texttt{0},\texttt{1}\}^\lambda$. We will calculate the probability that $\textsc{eavesdrop}(m)$ produces output c. That event happens only when

$$c = k \oplus m \iff k = m \oplus c.$$

The equivalence follows from the properties of xor given in Section 0.3. That is,

$$\Pr[\textsc{eavesdrop}(m) = c] = \Pr[k = m \oplus c],$$

where the probability is over uniform choice of $k \leftarrow \{\texttt{0},\texttt{1}\}^\lambda$.

We are considering a specific choice for m and c, so there is *only one* value of k that makes $k = m \oplus c$ true (causes m to encrypt to c), and that value is exactly $m \oplus c$. Since k is chosen *uniformly* from $\{\texttt{0},\texttt{1}\}^\lambda$, the probability of choosing the particular value $k = m \oplus c$ is $1/2^\lambda$.

In summary, for every m and c, the probability that $\textsc{eavesdrop}(m)$ outputs c is exactly $1/2^\lambda$. This means that the output of $\textsc{eavesdrop}(m)$, for any m, follows the uniform distribution. ■

One way to interpret this statement of security in more down-to-earth terms:

> If an attacker sees a *single* ciphertext, encrypted with one-time pad, where the key is chosen uniformly and kept secret from the attacker, then the ciphertext appears uniformly distributed.

Why is this significant? Taking the eavesdropper's point of view, suppose someone chooses a plaintext m and you get to see the resulting ciphertext — a sample from the distribution $\textsc{eavesdrop}(m)$. But this is a distribution that you can sample from yourself, even if you don't know m! You could have chosen a totally different m' and run $\textsc{eavesdrop}(m')$ in your imagination, and this would have produced the same distribution as $\textsc{eavesdrop}(m)$. The "real" ciphertext that you see *doesn't carry any information about* m if it is possible to sample from the same distribution without even knowing m!

Discussion

- **Isn't there a paradox?** Claim 1.2 says that c can always be decrypted to get m, but Claim 1.3 says that c contains no information about m! The answer to this riddle is that Claim 1.2 talks about what can be done with knowledge of the key k (Alice & Bob's perspective). Claim 1.3 talks about the output distribution of the EAVESDROP algorithm, which doesn't include k (Eve's perspective). In short, if you know k, then you can decrypt c to obtain m; if you don't know k, then c carries no information about m (in fact, it looks uniformly distributed). This is because m, c, k are all *correlated* in a delicate way.[3]

- **Isn't there another paradox?** Claim 1.3 says that the output of EAVESDROP(m) doesn't depend on m, but we can see the EAVESDROP algorithm literally using its argument m right there in the last line! The answer to this riddle is perhaps best illustrated by the previous illustrations of the EAVESDROP(010) and EAVESDROP(111) distributions. The two tables of values are indeed different (so the choice of $m \in \{010, 111\}$ clearly has some effect), but they represent the *same probability distribution* (since order doesn't matter). Claim 1.3 considers only the resulting probability distribution.

- You probably think about security in terms of a concrete "goal" for the attacker: recover the key, recover the plaintext, etc. Claim 1.3 doesn't really refer to attackers in that way, and it certainly doesn't specify a goal. Rather, we are thinking about security by comparing to some hypothetical "ideal" world. I would be satisfied if the attacker sees only a source of uniform bits, because in this hypothetical world there are no keys and no plaintexts to recover! Claim 1.3 says that when we actually use OTP, it looks just like this hypothetical world, from the attacker's point of view. If we imagine any "goal" at all for the attacker in this kind of reasoning, it's to detect that ciphertexts don't follow a uniform distribution. By showing that the attacker can't even achieve this modest goal, it shows that the attacker couldn't possibly achieve other, more natural, goals like key recovery and plaintext recovery.

Limitations

One-time pad is incredibly limited in practice. Most notably:

- Its keys are as long as the plaintexts they encrypt. This is basically unavoidable (see Exercise 2.11) and leads to a kind of chicken-and-egg dilemma in practice: If two users want to privately convey a λ-bit message, they first need to privately agree on a λ-bit string.

- A key can be used to encrypt only one plaintext (hence, "one-time" pad); see Exercise 1.6. Indeed, we can see that the EAVESDROP subroutine in Claim 1.3 provides no way for a caller to guarantee that two plaintexts are encrypted with the same key, so it is not clear how to use Claim 1.3 to argue about what happens in one-time pad when keys are intentionally reused in this way.

[3] This correlation is explored further in Chapter 3.

Despite these limitations, one-time pad illustrates fundamental ideas that appear in most forms of encryption in this course.

Questions to ponder upon for the reader.

Exercises

1.1. The one-time pad encryption of plaintext mario (when converted from ASCII to binary in the standard way) under key k is:

1000010000000111010101000001110000011101.

What is the one-time pad encryption of luigi under the same key?

1.2. Alice is using one-time pad and notices that when her key is the all-zeroes string $k = 0^\lambda$, then $\text{Enc}(k, m) = m$ and her message is sent in the clear! To avoid this problem, she decides to modify KeyGen to exclude the all-zeroes key. She modifies KeyGen to choose a key uniformly from $\{0,1\}^\lambda \setminus \{0^\lambda\}$, the set of all λ-bit strings except 0^λ. In this way, she guarantees that her plaintext is never sent in the clear.

Is it still true that the eavesdropper's ciphertext distribution is uniformly distributed on $\{0,1\}^\lambda$? Justify your answer.

May be more game playing exercises instead of this focus on OTP?

1.3. When Alice encrypts the key k itself using one-time pad, the ciphertext will always be the all-zeroes string! So if an eavesdropper sees the all-zeroes ciphertext, she learns that Alice encrypted the key itself. Does this contradict Claim 1.3? Why or why not?

1.4. What is so special about defining OTP using the XOR operation? Suppose we use the bitwise-AND operation (which we will write as '&') and define a variant of OTP as follows:

```
KeyGen:                  Enc(k, m ∈ {0,1}^λ):
  k ← {0,1}^λ              return k & m
  return k
```

Is this still a good choice for encryption? Why / why not?

1.5. Describe the flaw in this argument:

> Consider the following attack against one-time pad: upon seeing a ciphertext c, the eavesdropper tries every candidate key $k \in \{0,1\}^\lambda$ until she has found the one that was used, at which point she outputs the plaintext m. This contradicts the argument in Section 1.2 that the eavesdropper can obtain no information about m by seeing the ciphertext.

1.6. Suppose Alice encrypts two plaintexts m and m' using one-time pad with the same key k. What information about m and m' is leaked to an eavesdropper by doing this (assume the eavesdropper knows that Alice has reused k)? Be as specific as you can!

1.7. You (Eve) have intercepted two ciphertexts:

c_1 = 1111100101111001110011000001011110000110

$$c_2 = \texttt{1111101001100111110111010000100110001000}$$

You know that both are OTP ciphertexts, encrypted with the *same key*. You know that **either** c_1 is an encryption of `alpha` and c_2 is an encryption of `bravo` **or** c_1 is an encryption of `delta` and c_2 is an encryption of `gamma` (all converted to binary from ASCII in the standard way).

Which of these two possibilities is correct, and why? What was the key k?

1.8. A **known-plaintext attack** refers to a situation where an eavesdropper sees a ciphertext $c = \mathsf{Enc}(k, m)$ and also learns/knows what plaintext m was used to generate c.

(a) Show that a known-plaintext attack on OTP results in the attacker learning the key k.

(b) Can OTP be secure if it allows an attacker to recover the encryption key? Is this a contradiction to the security we showed for OTP? Explain.

1.9. Suppose we modify the subroutine discussed in Claim 1.3 so that it also returns k:

```
EAVESDROP'($m \in \{0,1\}^\lambda$):
    $k \leftarrow \{0,1\}^\lambda$
    $c := k \oplus m$
    return ($k$, $c$)
```

Is it still true that for every m, the output of EAVESDROP'(m) is distributed uniformly in $(\{0,1\}^\lambda)^2$? Or is the output distribution different for different choice of m?

1.10. In this problem we discuss the security of performing one-time pad encryption twice:

(a) Consider the following subroutine that models the result of applying one-time pad encryption with two *independent* keys:

```
EAVESDROP'($m \in \{0,1\}^\lambda$):
    $k_1 \leftarrow \{0,1\}^\lambda$
    $k_2 \leftarrow \{0,1\}^\lambda$
    $c := k_2 \oplus (k_1 \oplus m)$
    return $c$
```

Show that the output of this subroutine is uniformly distributed in $\{0,1\}^\lambda$.

(b) What security is provided by performing one-time pad encryption twice with *the same key*?

1.11. We mentioned that one-time pad keys can be used to encrypt only one plaintext, and how this was reflected in the EAVESDROP subroutine of Claim 1.3. Is there a similar restriction about re-using *plaintexts* in OTP (but with independently random keys for different ciphertexts)? If an eavesdropper *knows* that the same plaintext is encrypted twice (but doesn't know what the plaintext is), can she learn anything? Does Claim 1.3 have anything to say about a situation where the same plaintext is encrypted more than once?

1.12. There is nothing exclusively special about strings and XOR in OTP. We can get the same properties using integers mod n and addition mod n.

This problem considers a variant of one-time pad, in which the keys, plaintexts, and ciphertexts are all elements of $\mathbb{Z}_n$ instead of $\{0,1\}^\lambda$.

(a) What is the decryption algorithm that corresponds to the following encryption algorithm?

```
Enc($k, m \in \mathbb{Z}_n$):
  return $(k + m) \% n$
```
.

(b) Show that the output of the following subroutine is uniformly distributed in $\mathbb{Z}_n$:

```
EAVESDROP′($m \in \mathbb{Z}_n$):
  $k \leftarrow \mathbb{Z}_n$
  $c := (k + m) \% n$
  return $c$
```
.

(c) It's not just the distribution of keys that is important. The way that the key is combined with the plaintext is also important. Show that the output of the following subroutine is **not** necessarily uniformly distributed in $\mathbb{Z}_n$:

```
EAVESDROP′($m \in \mathbb{Z}_n$):
  $k \leftarrow \mathbb{Z}_n$
  $c := (k \cdot m) \% n$
  return $c$
```
.