Cryptography

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Cryptography

### 

* Definition of perfect secrecy
* The one-time pad encryption scheme
* Necessity of long keys for perfect secrecy
* Computational secrecy and the derandomized one-time pad.
* Public key encryption
* A taste of advanced topics

*“Human ingenuity cannot concoct a cipher which human ingenuity cannot resolve.”*, Edgar Allen Poe, 1841

*“A good disguise should not reveal the person’s height”*, Shafi Goldwasser and Silvio Micali, 1982

*““Perfect Secrecy” is defined by requiring of a system that after a cryptogram is intercepted by the enemy the a posteriori probabilities of this cryptogram representing various messages be identically the same as the a priori probabilities of the same messages before the interception. It is shown that perfect secrecy is possible but requires, if the number of messages is finite, the same number of possible keys."*, Claude Shannon, 1945

*“We stand today on the brink of a revolution in cryptography.”*, Whitfeld Diffie and Martin Hellman, 1976

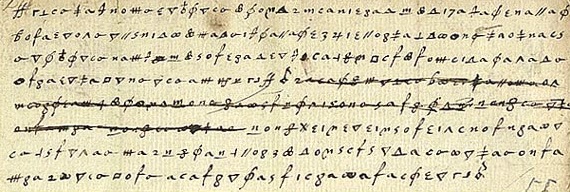
Cryptography - the art or science of “secret writing” - has been around for several millennia, and for almost all of that time Edgar Allan Poe’s quote above held true. Indeed, the history of cryptography is littered with the figurative corpses of cryptosystems believed secure and then broken, and sometimes with the actual corpses of those who have mistakenly placed their faith in these cryptosystems.

Yet, something changed in the last few decades, which is the “revolution” alluded to (and to a large extent initiated by) Diffie and Hellman’s 1976 paper quoted above. New cryptosystems have been found that have not been broken despite being subjected to immense efforts involving both human ingenuity and computational power on a scale that completely dwarves the “code breakers” of Poe’s time. Even more amazingly, these cryptosystem are not only seemingly unbreakable, but they also achieve this under much harsher conditions. Not only do today’s attackers have more computational power but they also have more data to work with. In Poe’s age, an attacker would be lucky if they got access to more than a few encryptions of known messages. These days attackers might have massive amounts of data- terabytes or more - at their disposal. In fact, with *public key* encryption, an attacker can generate as many ciphertexts as they wish.

The key to this success has been a clearer understanding of both how to *define* security for cryptographic tools and how to relate this security to *concrete computational problems*. Cryptography is a vast and continuously changing topic, but we will touch on some of these issues in this chapter.

## Classical cryptosystems

A great many cryptosystems have been devised and broken throughout the ages. Let us recount just some of these stories. In 1587, Mary the queen of Scots, and the heir to the throne of England, wanted to arrange the assassination of her cousin, queen Elisabeth I of England, so that she could ascend to the throne and finally escape the house arrest under which she had been for the last 18 years. As part of this complicated plot, she sent a coded letter to Sir Anthony Babington.



Snippet from encrypted communication between queen Mary and Sir Babington

Mary used what’s known as a *substitution cipher* where each letter is transformed into a different obscure symbol (see maryscottletterfig). At a first look, such a letter might seem rather inscrutable- a meaningless sequence of strange symbols. However, after some thought, one might recognize that these symbols *repeat* several times and moreover that different symbols repeat with different frequencies. Now it doesn’t take a large leap of faith to assume that perhaps each symbol corresponds to a different letter and the more frequent symbols correspond to letters that occur in the alphabet with higher frequency. From this observation, there is a short gap to completely breaking the cipher, which was in fact done by queen Elisabeth’s spies who used the decoded letters to learn of all the co-conspirators and to convict queen Mary of treason, a crime for which she was executed. Trusting in superficial security measures (such as using “inscrutable” symbols) is a trap that users of cryptography have been falling into again and again over the years. (As in many things, this is the subject of a great XKCD cartoon, see XKCDnavajofig.)



XKCD’s take on the added security of using uncommon symbols

The [Vigenère cipher](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher) is named after Blaise de Vigenère who described it in a book in 1586 (though it was invented earlier by Bellaso). The idea is to use a collection of substitution cyphers - if there are different ciphers then the first letter of the plaintext is encoded with the first cipher, the second with the second cipher, the with the cipher, and then the letter is again encoded with the first cipher. The key is usually a word or a phrase of letters, and the substitution cipher is obtained by shifting each letter positions in the alphabet. This “flattens” the frequencies and makes it much harder to do frequency analysis, which is why this cipher was considered “unbreakable” for 300+ years and got the nickname “le chiffre indéchiffrable” (“the unbreakable cipher”). Nevertheless, Charles Babbage cracked the Vigenère cipher in 1854 (though he did not publish it). In 1863 Friedrich Kasiski broke the cipher and published the result. The idea is that once you guess the length of the cipher, you can reduce the task to breaking a simple substitution cipher which can be done via frequency analysis (can you see why?). Confederate generals used Vigenère regularly during the civil war, and their messages were routinely cryptanalzed by Union officers.



Confederate Cipher Disk for implementing the Vigenère cipher



Confederate encryption of the message “Gen’l Pemberton: You can expect no help from this side of the river. Let Gen’l Johnston know, if possible, when you can attack the same point on the enemy’s lines. Inform me also and I will endeavor to make a diversion. I have sent some caps. I subjoin a despatch from General Johnston.”

The *Enigma* cipher was a mechanical cipher (looking like a typewriter, see enigmafig) where each letter typed would get mapped into a different letter depending on the (rather complicated) key and current state of the machine which had several rotors that rotated at different paces. An identically wired machine at the other end could be used to decrypt. Just as many ciphers in history, this has also been believed by the Germans to be “impossible to break” and even quite late in the war they refused to believe it was broken despite mounting evidence to that effect. (In fact, some German generals refused to believe it was broken even *after* the war.) Breaking Enigma was an heroic effort which was initiated by the Poles and then completed by the British at Bletchley Park, with Alan Turing (of the Turing machines) playing a key role. As part of this effort the Brits built arguably the world’s first large scale mechanical computation devices (though they looked more similar to washing machines than to iPhones). They were also helped along the way by some quirks and errors of the German operators. For example, the fact that their messages ended with “Heil Hitler” turned out to be quite useful.



In the *Enigma* mechanical cipher the secret key would be the settings of the rotors and internal wires. As the operator types up their message, the encrypted version appeared in the display area above, and the internal state of the cipher was updated (and so typing the same letter twice would generally result in two different letters output). Decrypting follows the same process: if the sender and receiver are using the same key then typing the ciphertext would result in the plaintext appearing in the display.

Here is one entertaining anecdote: the Enigma machine would never map a letter to itself. In March 1941, Mavis Batey, a cryptanalyst at Bletchley Park received a very long message that she tried to decrypt. She then noticed a curious property— the message did *not* contain the letter “L”.[[1]](#footnote-35) She realized that the probability that no “L”’s appeared in the message is too small for this to happen by chance. Hence she surmised that the original message must have been composed *only* of L’s. That is, it must have been the case that the operator, perhaps to test the machine, have simply sent out a message where he repeatedly pressed the letter “L”. This observation helped her decode the next message, which helped inform of a planned Italian attack and secure a resounding British victory in what became known as “the Battle of Cape Matapan”. Mavis also helped break another Enigma machine. Using the information she provided, the Brits were able to feed the Germans with the false information that the main allied invasion would take place in Pas de Calais rather than on Normandy.

In the words of General Eisenhower, the intelligence from Bletchley park was of “priceless value”. It made a huge difference for the Allied war effort, thereby shortening World War II and saving millions of lives. See also [this interview with Sir Harry Hinsley](http://www.cix.co.uk/~klockstone/hinsley.htm).

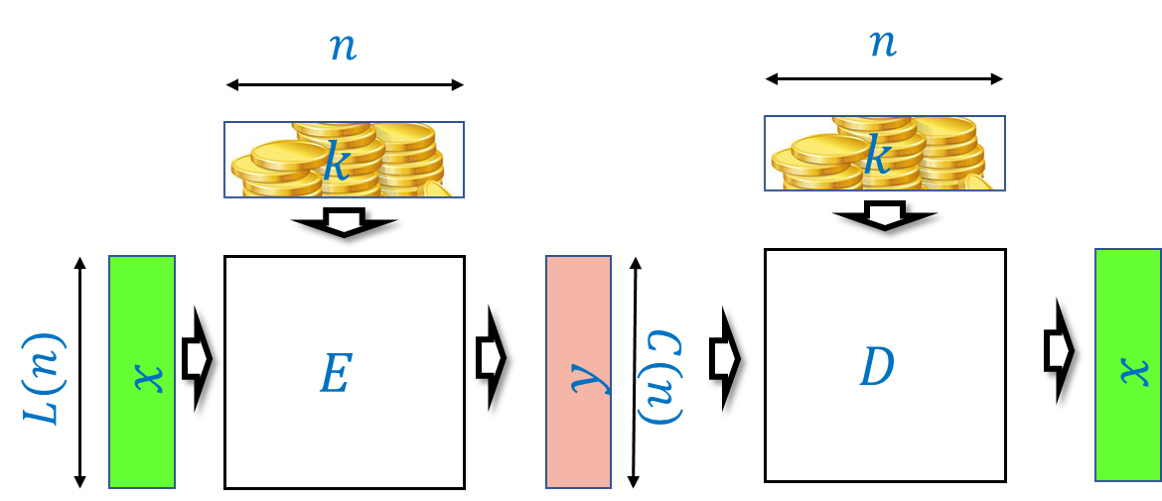
## Defining encryption

Many of the troubles that cryptosystem designers faced over history (and still face!) can be attributed to not properly defining or understanding what are the goals they want to achieve in the first place. Let us focus on the setting of *private key encryption*. (This is also known as “symmetric encryption”; for thousands of years, “private key encryption” was synonymous with encryption and only in the 1970’s was the concept of *public key encryption* invented, see publickeyencdef.) A *sender* (traditionally called “Alice”) wants to send a message (known also as a *plaintext*) to a *receiver* (traditionally called “Bob”). They would like their message to be kept secret from an *adversary* who listens in or “eavesdrops” on the communication channel (and is traditionally called “Eve”).

Alice and Bob share a *secret key* . (While the letter is often used elsewhere in the book to denote a natural number, in this chapter we use it to denote the string corresponding to a secret key.) Alice uses the key to “scramble” or *encrypt* the plaintext into a *ciphertext* , and Bob uses the key to “unscramble” or *decrypt* the ciphertext back into the plaintext . This motivates the following definition which attempts to capture what it means for an encryption scheme to be *valid* or “make sense”, regardless of whether or not it is *secure*:

Let and be two functions mapping natural numbers to natural numbers. A pair of polynomial-time computable functions mapping strings to strings is a *valid private key encryption scheme* (or *encryption scheme* for short) with plaintext length function and ciphertext length function if for every , and , and

We will often write the first input (i.e., the key) to the encryption and decryption as a subscript and so can write eqvalidenc also as .



A private-key encryption scheme is a pair of algorithms such that for every key and plaintext , is a ciphertext of length . The encryption scheme is *valid* if for every such , . That is, the decryption of an encryption of is , as long as both encryption and decryption use the same key.

Prove that for every valid encryption scheme with functions . for every .

For every fixed key , the equation eqvalidenc implies that the map inverts the map , which in particular means that the map must be one to one. Hence its codomain must be at least as large as its domain, and since its domain is and its codomain is it follows that .

Since the ciphertext length is always at least the plaintext length (and in most applications it is not much longer than that), we typically focus on the plaintext length as the quantity to optimize in an encryption scheme. The *larger* is, the better the scheme, since it means we need a shorter secret key to protect messages of the same length.

## Defining security of encryption

encryptiondef says nothing about the *security* of and , and even allows the trivial encryption scheme that ignores the key altogether and sets for every . Defining security is not a trivial matter.

### 

You would appreciate the subtleties of defining security of encryption more if at this point you take a five minute break from reading, and try (possibly with a partner) to brainstorm on how you would mathematically define the notion that an encryption scheme is *secure*, in the sense that it protects the secrecy of the plaintext .

Throughout history, many attacks on cryptosystems were rooted in the cryptosystem designers’ reliance on “security through obscurity”— trusting that the fact their *methods* are not known to their enemy will protect them from being broken. This is a faulty assumption - if you reuse a method again and again (even with a different key each time) then eventually your adversaries will figure out what you are doing. And if Alice and Bob meet frequently in a secure location to decide on a new method, they might as well take the opportunity to exchange their secrets. These considerations led Auguste Kerckhoffs in 1883 to state the following principle:

*A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.*[[2]](#footnote-44)

Why is it OK to assume the key is secret and not the algorithm? Because we can always choose a fresh key. But of course that won’t help us much if our key is “1234” or “passw0rd!”. In fact, if you use *any* deterministic algorithm to choose the key then eventually your adversary will figure this out. Therefore for security we must choose the key at *random* and can restate Kerckhoffs’s principle as follows:

*There is no secrecy without randomness*

This is such a crucial point that is worth repeating:

There is no *secrecy* without *randomness*.

At the heart of every cryptographic scheme there is a secret key, and the secret key is always chosen at random. A corollary of that is that to understand cryptography, you need to know probability theory.

Choosing the secrets for cryptography requires generating randomness, which is often done by measuring some “unpredictable” or “high entropy” data, and then applying hash functions to the result to “extract” a uniformly random string. Great care must be taken in doing this, and randomness generators often turn out to be the Achilles heel of secure systems.

In 2006 a programmer removed a line of code from the procedure to generate entropy in OpenSSL package distributed by Debian since it caused a warning in some automatic verification code. As a result for two years (until this was discovered) all the randomness generated by this procedure used only the process ID as an “unpredictable” source. This means that all communication done by users in that period is fairly easily breakable (and in particular, if some entities recorded that communication they could break it also retroactively). See [XKCD’s take](http://www.xkcd.com/424/) on that incident.

In 2012 two separate teams of researchers scanned a large number of RSA keys on the web and found out that about 4 percent of them are easy to break. The main issue were devices such as routers, internet-connected printers and such. These devices sometimes run variants of Linux- a desktop operating system- but without a hard drive, mouse or keyboard, they don’t have access to many of the entropy sources that desktop have. Coupled with some good old fashioned ignorance of cryptography and software bugs, this led to many keys that are downright trivial to break, see [this blog post](https://freedom-to-tinker.com/blog/nadiah/new-research-theres-no-need-panic-over-factorable-keys-just-mind-your-ps-and-qs/) and [this web page](https://factorable.net/) for more details.

Since randomness is so crucial to security, breaking the procedure to generate randomness can lead to a complete break of the system that uses this randomness. Indeed, the Snowden documents, combined with observations of Shumow and Ferguson, [strongly suggest](https://en.wikipedia.org/wiki/Dual_EC_DRBG) that the NSA has deliberately inserted a *trapdoor* in one of the pseudorandom generators published by the National Institute of Standards and Technologies (NIST). Fortunately, this generator wasn’t widely adapted but apparently the NSA did pay 10 million dollars to RSA security so the latter would make this generator their default option in their products.

## Perfect secrecy

If you think about encryption scheme security for a while, you might come up with the following principle for defining security: *“An encryption scheme is secure if it is not possible to recover the key from ”*. However, a moment’s thought shows that the key is not really what we’re trying to protect. After all, the whole point of an encryption is to protect the confidentiality of the *plaintext* . So, we can try to define that *“an encryption scheme is secure if it is not possible to recover the plaintext from ”*. Yet it is not clear what this means either. Suppose that an encryption scheme reveals the first 10 bits of the plaintext . It might still not be possible to recover completely, but on an intuitive level, this seems like it would be extremely unwise to use such an encryption scheme in practice. Indeed, often even *partial information* about the plaintext is enough for the adversary to achieve its goals.

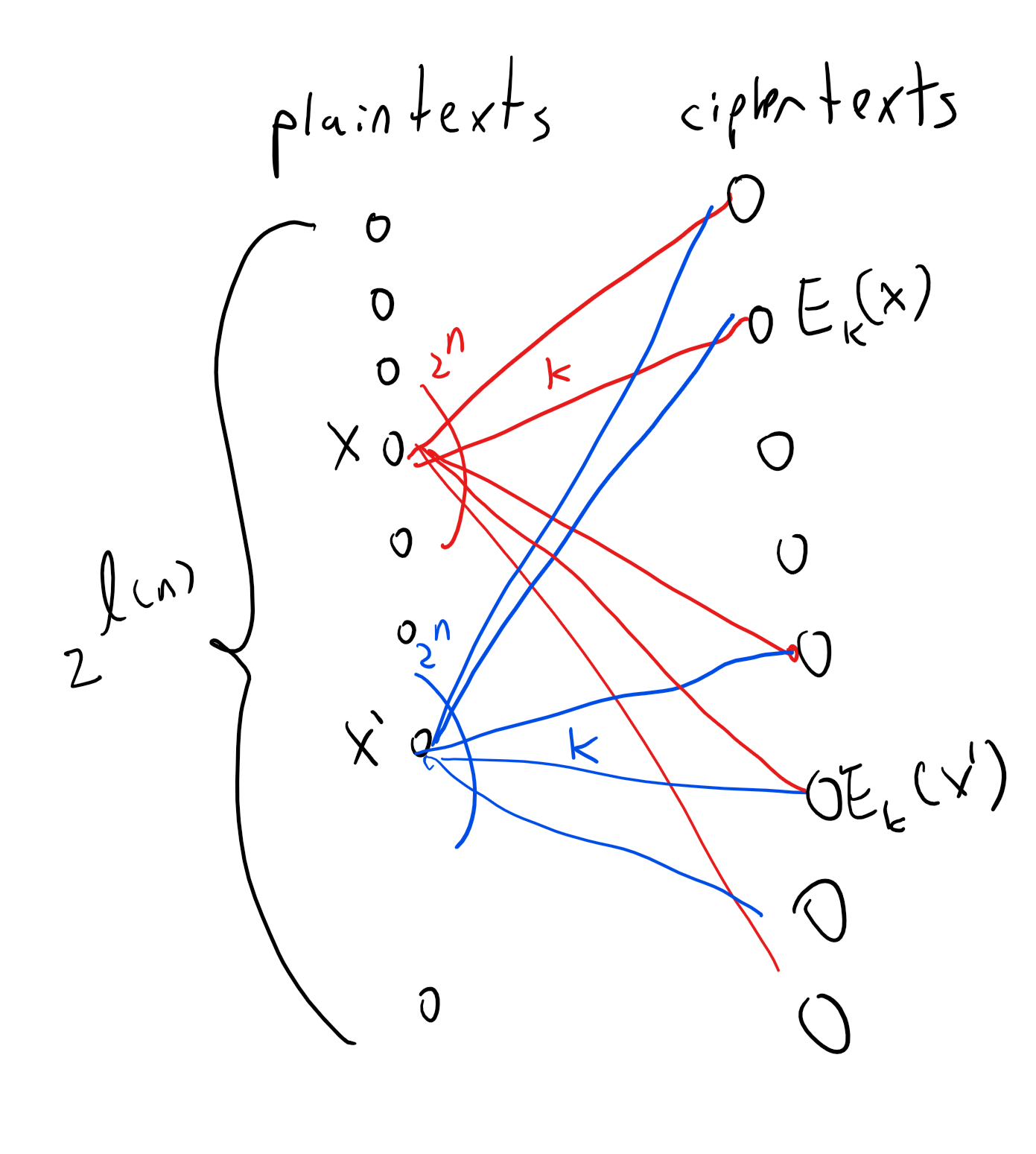
The above thinking led Shannon in 1945 to formalize the notion of *perfect secrecy*, which is that an encryption reveals absolutely nothing about the message. There are several equivalent ways to define it, but perhaps the cleanest one is the following:

A valid encryption scheme with plaintext length is *perfectly secret* if for every and plaintexts , the following two distributions and over are identical:

* is obtained by sampling and outputting .
* is obtained by sampling and outputting .

This definition might take more than one reading to parse. Try to think of how this condition would correspond to your intuitive notion of “learning no information” about from observing , and to Shannon’s quote in the beginning of this chapter.

In particular, suppose that you knew ahead of time that Alice sent either an encryption of or an encryption of . Would you learn anything new from observing the encryption of the message that Alice actually sent? It may help you to look at perfectsecfig.



For any key length , we can visualize an encryption scheme as a graph with a vertex for every one of the possible plaintexts and for every one of the ciphertexts in of the form for and . For every plaintext and key , we add an edge labeled between and . By the validity condition, if we pick any fixed key , the map must be one-to-one. The condition of perfect secrecy simply corresponds to requiring that every two plaintexts and have exactly the same set of neighbors (or multi-set, if there are parallel edges).

### Example: Perfect secrecy in the battlefield

To understand perfectsecrecy, suppose that Alice sends only one of two possible messages: “attack” or “retreat”, which we denote by and respectively, and that she sends each one of those messages with probability . Let us put ourselves in the shoes of *Eve*, the eavesdropping adversary. A priori we would have guessed that Alice sent either or with probability . Now we observe where is a uniformly chosen key in . How does this new information cause us to update our beliefs on whether Alice sent the plaintext or the plaintext ?

### 

Before reading the next paragraph, you might want to try the analysis yourself. You may find it useful to look at the [Wikipedia entry on Bayesian Inference](https://en.wikipedia.org/wiki/Bayesian_inference) or [these MIT lecture notes](https://ocw.mit.edu/courses/mathematics/18-05-introduction-to-probability-and-statistics-spring-2014/readings/MIT18_05S14_Reading11.pdf).

Let us define to be the probability (taken over ) that and similarly to be . Note that, since Alice chooses the message to send at random, our a priori probability for observing is . However, as per perfectsecrecy, the perfect secrecy condition guarantees that ! Let us denote the number by . By the formula for conditional probability, the probability that Alice sent the message conditioned on our observation is simply

(The equation bayeseq is a special case of *Bayes’ rule* which, although a simple restatement of the formula for conditional probability, is an extremely important and widely used tool in statistics and data analysis.)

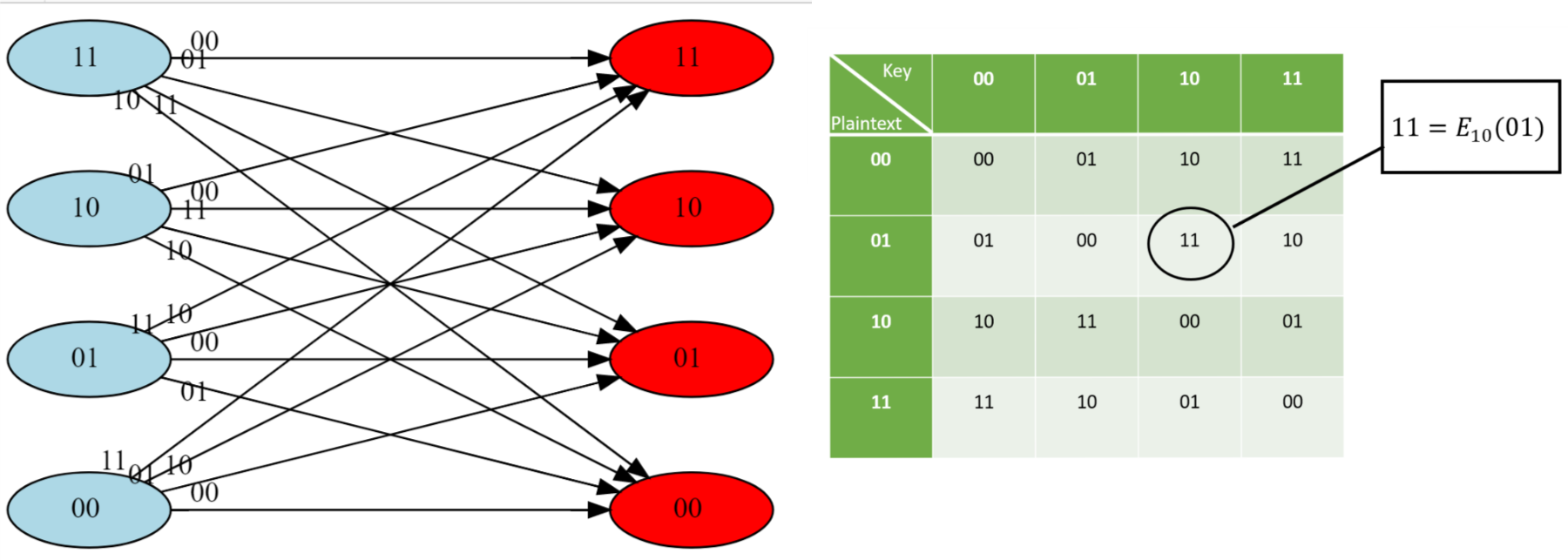
Since the probability that and is the ciphertext is equal to , and the a priori probability of observing is , we can rewrite bayeseq as

using the fact that . This means that observing the ciphertext did not help us at all! We still would not be able to guess whether Alice sent “attack” or “retreat” with better than 50/50 odds!

This example can be vastly generalized to show that perfect secrecy is indeed “perfect” in the sense that observing a ciphertext gives Eve *no additional information* about the plaintext beyond her a priori knowledge.

### Constructing perfectly secret encryption

*Perfect secrecy* is an extremely strong condition, and implies that an eavesdropper does not learn *any* information from observing the ciphertext. You might think that an encryption scheme satisfying such a strong condition will be impossible, or at least extremely complicated, to achieve. However it turns out we can in fact obtain perfectly secret encryption scheme fairly easily. Such a scheme for two-bit messages is illustrated in onetimepadtwofig



A perfectly secret encryption scheme for two-bit keys and messages. The blue vertices represent plaintexts and the red vertices represent ciphertexts, each edge mapping a plaintext to a ciphertext is labeled with the corresponding key . Since there are four possible keys, the degree of the graph is four and it is in fact a complete bipartite graph. The encryption scheme is valid in the sense that for every , the map is one-to-one, which in other words means that the set of edges labeled with is a *matching*.

In fact, this can be generalized to any number of bits:

### 

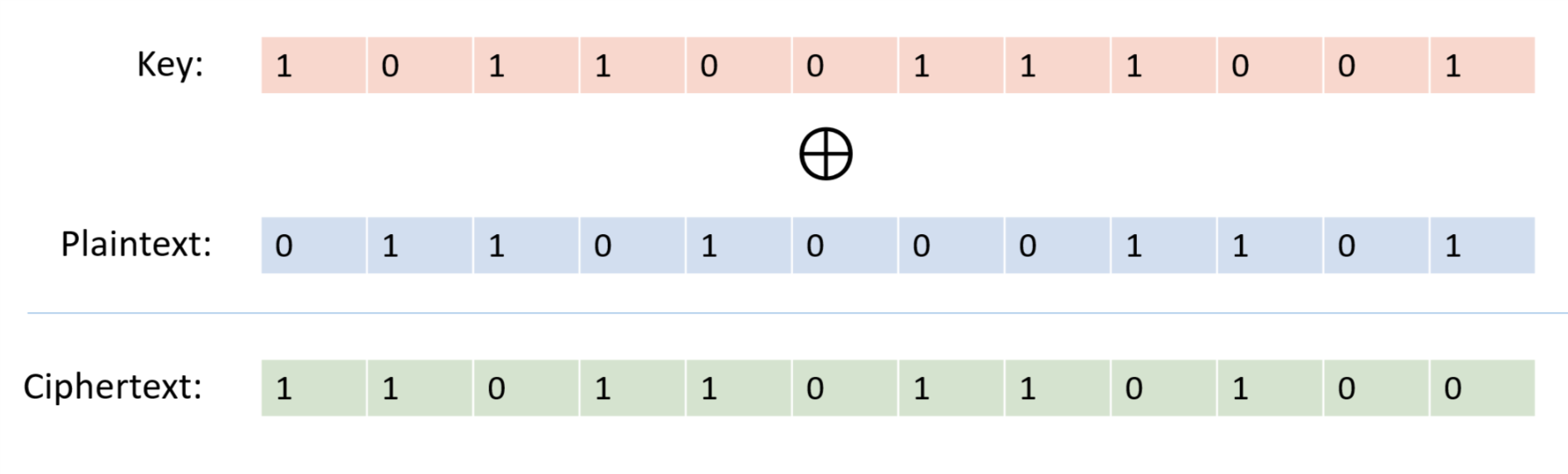
There is a perfectly secret valid encryption scheme with .

### 

Our scheme is the [one-time pad](https://en.wikipedia.org/wiki/One-time_pad) also known as the “Vernam Cipher”, see onetimepadfig. The encryption is exceedingly simple: to encrypt a message with a key we simply output where is the bitwise XOR operation that outputs the string corresponding to XORing each coordinate of and .

For two binary strings and of the same length , we define to be the string such that for every . The encryption scheme is defined as follows: and . By the associative law of addition (which works also modulo two), , using the fact that for every bit , and . Hence form a valid encryption.

To analyze the perfect secrecy property, we claim that for every , the distribution where is simply the uniform distribution over , and hence in particular the distributions and are identical for every . Indeed, for every particular , the value is output by if and only if which holds if and only if . Since is chosen uniformly at random in , the probability that happens to equal is exactly , which means that every string is output by with probability .



In the *one time pad* encryption scheme we encrypt a plaintext with a key by the ciphertext where denotes the bitwise XOR operation.

### 

The argument above is quite simple but is worth reading again. To understand why the one-time pad is perfectly secret, it is useful to envision it as a bipartite graph as we’ve done in onetimepadtwofig. (In fact the encryption scheme of onetimepadtwofig is precisely the one-time pad for .) For every , the one-time pad encryption scheme corresponds to a bipartite graph with vertices on the “left side” corresponding to the plaintexts in and vertices on the “right side” corresponding to the ciphertexts . For every and , we connect to the vertex with an edge that we label with . One can see that this is the complete bipartite graph, where every vertex on the left is connected to *all* vertices on the right. In particular this means that for every left vertex , the distribution on the ciphertexts obtained by taking a random and going to the neighbor of on the edge labeled is the uniform distribution over . This ensures the perfect secrecy condition.

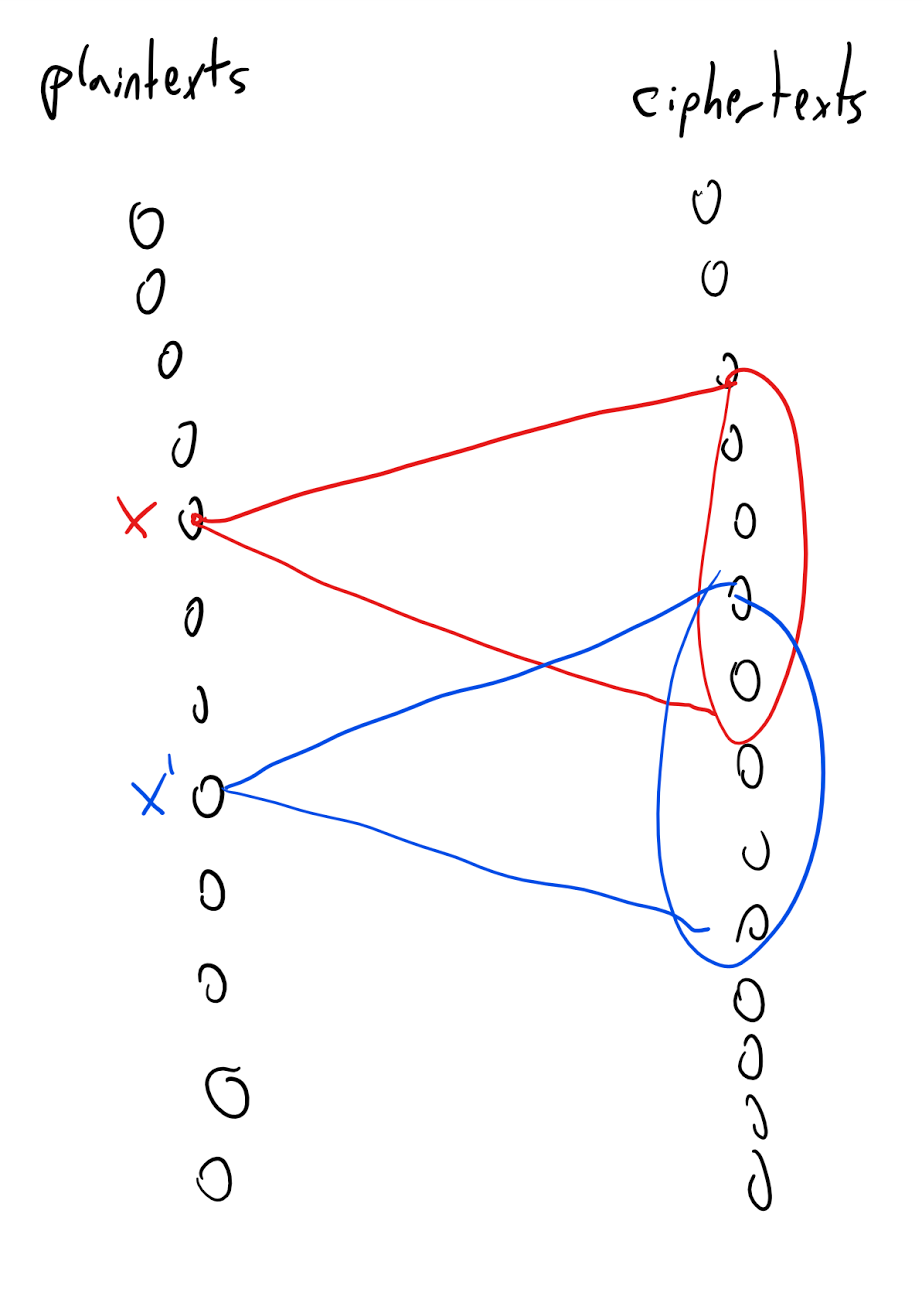
## Necessity of long keys

So, does onetimepad give the final word on cryptography, and means that we can all communicate with perfect secrecy and live happily ever after? No it doesn’t. While the one-time pad is efficient, and gives perfect secrecy, it has one glaring disadvantage: to communicate bits you need to store a key of length . In contrast, practically used cryptosystems such as AES-128 have a short key of bits (i.e., bytes) that can be used to protect terabytes or more of communication! Imagine that we all needed to use the one time pad. If that was the case, then if you had to communicate with people, you would have to maintain (securely!) huge files that are each as long as the length of the maximum total communication you expect with that person. Imagine that every time you opened an account with Amazon, Google, or any other service, they would need to send you in the mail (ideally with a secure courier) a DVD full of random numbers, and every time you suspected a virus, you’d need to ask all these services for a fresh DVD. This doesn’t sound so appealing.

This is not just a theoretical issue. The Soviets have used the one-time pad for their confidential communication since before the 1940’s. In fact, even before Shannon’s work, the U.S. intelligence already knew in 1941 that the one-time pad is in principle “unbreakable” (see page 32 in the [Venona document](http://nsarchive.gwu.edu/NSAEBB/NSAEBB278/01.PDF)). However, it turned out that the hassle of manufacturing so many keys for all the communication took its toll on the Soviets and they ended up reusing the same keys for more than one message. They did try to use them for completely different receivers in the (false) hope that this wouldn’t be detected. The [Venona Project](https://en.wikipedia.org/wiki/Venona_project) of the U.S. Army was founded in February 1943 by Gene Grabeel (see genegrabeelfig), a former home economics teacher from Madison Heights, Virgnia and Lt. Leonard Zubko. In October 1943, they had their breakthrough when it was discovered that the Russians were reusing their keys. In the 37 years of its existence, the project has resulted in a treasure chest of intelligence, exposing hundreds of KGB agents and Russian spies in the U.S. and other countries, including Julius Rosenberg, Harry Gold, Klaus Fuchs, Alger Hiss, Harry Dexter White and many others.



Gene Grabeel, who founded the U.S. Russian SigInt program on 1 Feb 1943. Photo taken in 1942, see Page 7 in the Venona historical study.



An encryption scheme where the number of keys is smaller than the number of plaintexts corresponds to a bipartite graph where the degree is smaller than the number of vertices on the left side. Together with the validity condition this implies that there will be two left vertices with non-identical neighborhoods, and hence the scheme does *not* satisfy perfect secrecy.

Unfortunately it turns out that that such long keys are *necessary* for perfect secrecy:

### 

For every perfectly secret encryption scheme the length function satisfies .

### 

The idea behind the proof is illustrated in longkeygraphfig. We define a graph between the plaintexts and ciphertexts, where we put an edge between plaintext and ciphertext if there is some key such that . The *degree* of this graph is at most the number of potential keys. The fact that the degree is smaller than the number of plaintexts (and hence of ciphertexts) implies that there would be two plaintexts and with different sets of neighbors, and hence the distribution of a ciphertext corresponding to (with a random key) will not be identical to the distribution of a ciphertext corresponding to .

Let be a valid encryption scheme with messages of length and key of length . We will show that is not perfectly secret by providing two plaintexts such that the distributions and are not identical, where is the distribution obtained by picking and outputting .

We choose . Let be the set of all ciphertexts that have nonzero probability of being output in . That is, . Since there are only keys, we know that .

We will show the following claim:

**Claim I:** There exists some and such that .

Claim I implies that the string has positive probability of being output by and zero probability of being output by and hence in particular and are not identical. To prove Claim I, just choose a fixed . By the validity condition, the map is a one to one map of to and hence in particular the *image* of this map which is the set has size at least (in fact exactly) . Since , this means that and so in particular there exists some string in . But by the definition of this means that there is some such that which concludes the proof of Claim I and hence of longkeysthm.

## Computational secrecy

To sum up the previous episodes, we now know that:

* It is possible to obtain a perfectly secret encryption scheme with key length the same as the plaintext.

and

* It is not possible to obtain such a scheme with key that is even a single bit shorter than the plaintext.

How does this mesh with the fact that, as we’ve already seen, people routinely use cryptosystems with a 16 byte (i.e., 128 bit) key but many terabytes of plaintext? The proof of longkeysthm does give in fact a way to break all these cryptosystems, but an examination of this proof shows that it only yields an algorithm with time *exponential in the length of the key*. This motivates the following relaxation of perfect secrecy to a condition known as *“computational secrecy”*. Intuitively, an encryption scheme is computationally secret if no polynomial time algorithm can break it. The formal definition is below:

Let be a valid encryption scheme where for keys of length , the plaintexts are of length and the ciphertexts are of length . We say that is *computationally secret* if for every polynomial , and large enough , if is an -input and single output NAND-CIRC program of at most lines, and then

### 

compsecdef requires a second or third read and some practice to truly understand. One excellent exercise to make sure you follow it is to see that if we allow to be an *arbitrary* function mapping to , and we replace the condition in eqindist that the lefthand side is smaller than with the condition that it is equal to then we get the perfect secrecy condition of perfectsecrecy. Indeed if the distributions and are identical then applying any function to them we get the same expectation. On the other hand, if the two distributions above give a different probability for some element , then the function that outputs iff will have a different expectation under the former distribution than under the latter.

compsecdef raises two natural questions:

* Is it strong enough to ensure that a computationally secret encryption scheme protects the secrecy of messages that are encrypted with it?
* It is weak enough that, unlike perfect secrecy, it is possible to obtain a computationally secret encryption scheme where the key is much smaller than the message?

To the best of our knowledge, the answer to both questions is *Yes*.

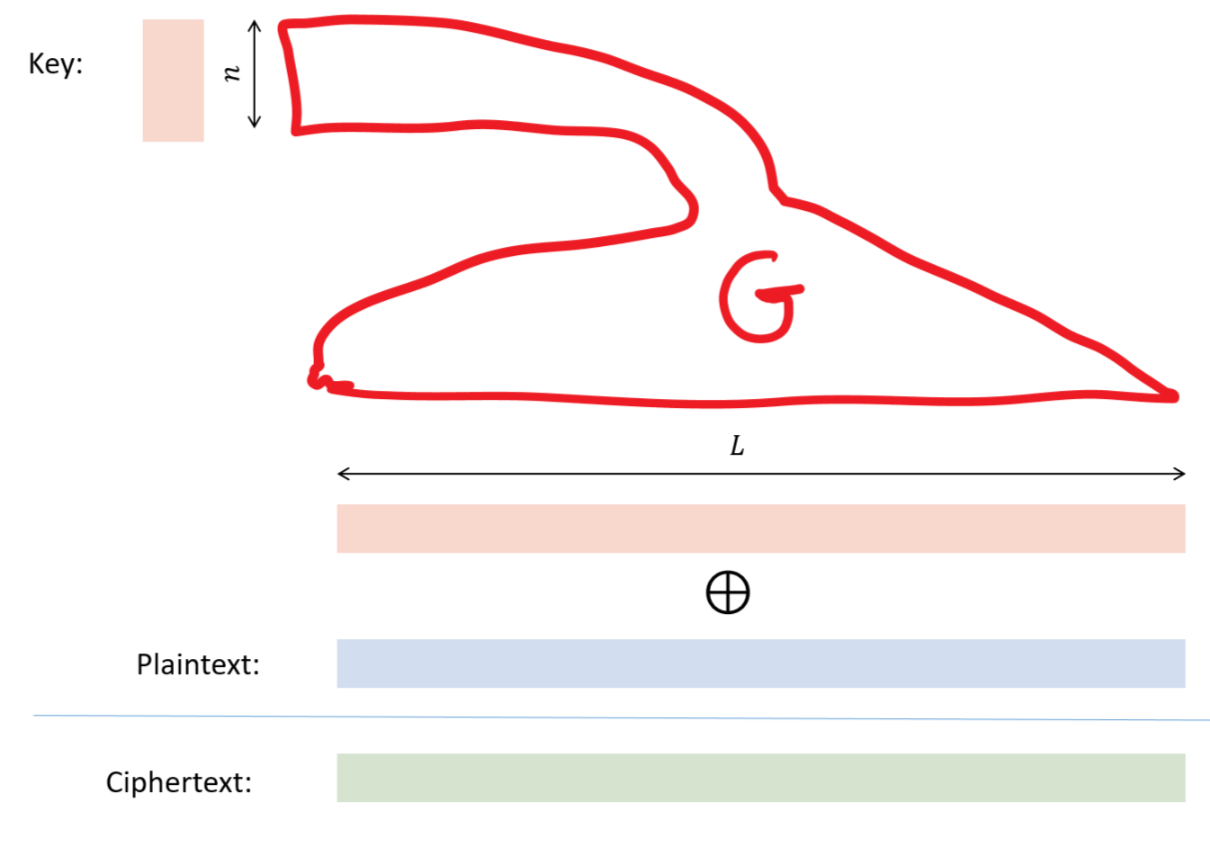
*Computational hardness* is necessary and sufficient for a great many cryptographic constructions, and in particular for encryption schemes with keys shorter than the message.

Regarding the first question, it is not hard to show that if, for example, Alice uses a computationally secret encryption algorithm to encrypt either “attack” or “retreat” (each chosen with probability ), then as long as she’s restricted to polynomial-time algorithms, an adversary Eve will not be able to guess the message with probability better than, say, , even after observing its encrypted form. (We omit the proof, but it is an excellent exercise for you to work it out on your own.)

To answer the second question we will show that under the same assumption we used for derandomizing , we can obtain a computationally secret cryptosystem where the key is almost *exponentially* smaller than the plaintext.

### Stream ciphers or the “derandomized one-time pad”

It turns out that if pseudorandom generators exist as in the optimal PRG conjecture, then there exists a computationally secret encryption scheme with keys that are much shorter than the plaintext. The construction below is known as a [stream cipher](https://en.wikipedia.org/wiki/Stream_cipher), though perhaps a better name is the “derandomized one-time pad”. It is widely used in practice with keys on the order of a few tens or hundreds of bits protecting many terabytes or even petabytes of communication.



In a *stream cipher* or “derandomized one-time pad” we use a pseudorandom generator to obtain an encryption scheme with a key length of and plaintexts of length . We encrypt the plaintext with key by the ciphertext .

We start by recalling the notion of a *pseudorandom generator*, as defined in prgdef. For this chapter, we will fix a special case of the definition:

Let be some function. A *cryptographic pseudorandom generator* with stretch is a polynomial-time computable function such that:

* For every and , .
* For every polynomial and large enough, if is a circuit of inputs, one output, and at most gates then

In this chapter we will call a cryptographic pseudorandom generator simply a *pseudorandom generator* or PRG for short. The optimal PRF conjecture of optimalprgconj implies that there is a pseudorandom generator that can “fool” circuits of *exponential size* and where the gap in probabilities is at most one over an exponential quantity. Since exponential grow faster than every polynomial, the optimal PRG conjecture implies the following:

**The crypto PRG conjecture:** For every , there is a cryptographic pseudorandom generator with .

The crypto PRG conjecture is a weaker conjecture than the optimal PRG conjecture, but it too (as we will see) is still stronger than the conjecture that .

### 

Suppose that the crypto PRG conjecture is true. Then for every constant there is a computationally secret encryption scheme with plaintext length at least .

### 

The proof is illustrated in derandonetimepadfig. We simply take the one-time pad on bit plaintexts, but replace the key with where is a string in and is a pseudorandom generator. Since the one time pad cannot be broken, an adversary that breaks the derandomized one-time pad can be used to distinguish between the output of the pseudorandom generator and the uniform distribution.

Let for be the restriction to input length of the pseudorandom generator whose existence we are guaranteed from the crypto PRG conjecture. We now define our encryption scheme as follows: given key and plaintext , the encryption is simply . To decrypt a string we output . This is a valid encryption since is computable in polynomial time and for every .

Computational secrecy follows from the condition of a pseudorandom generator. Suppose, towards a contradiction, that there is a polynomial , NAND-CIRC program of at most lines and such that

(We use here the simple fact that for a -valued random variable , .)

By the definition of our encryption scheme, this means that

Now since (as we saw in the security analysis of the one-time pad), for every strings , the distribution and are identical, where . Hence

By plugging eqprgsectwo into eqprgsecone we can derive that

(Please make sure that you can see why this is true.)

Now we can use the *triangle inequality* that for every two numbers , applying it for and to derive

In particular, either the first term or the second term of the lefthand-side of eqprgsefour must be at least . Let us assume the first case holds (the second case is analyzed in exactly the same way). Then we get that

But if we now define the NAND-CIRC program that on input outputs then (since XOR of bits can be computed in lines), we get that has lines and by distingprgeq it can distinguish between an input of the form and an input of the form with advantage better than . Since a polynomial is dominated by an exponential, if we make large enough, this will contradict the security of the pseudorandom generator .

### 

The two most widely used forms of (private key) encryption schemes in practice are *stream ciphers* and *block ciphers*. (To make things more confusing, a block cipher is always used in some [mode of operation](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation) and some of these modes effectively turn a block cipher into a stream cipher.) A block cipher can be thought as a sort of a “random invertible map” from to , and can be used to construct a pseudorandom generator and from it a stream cipher, or to encrypt data directly using other modes of operations. There are a great many other security notions and considerations for encryption schemes beyond computational secrecy. Many of those involve handling scenarios such as *chosen plaintext*, *man in the middle*, and *chosen ciphertext* attacks, where the adversary is not just merely a passive eavesdropper but can influence the communication in some way. While this chapter is meant to give you some taste of the ideas behind cryptography, there is much more to know before applying it correctly to obtain secure applications, and a great many people have managed to get it wrong.

## Computational secrecy and

We’ve also mentioned before that an efficient algorithm for could be used to break all cryptography. We now give an example of how this can be done:

If then there is no computationally secret encryption scheme with .

Furthermore, for every valid encryption scheme with there is a polynomial such that for every large enough there exist and a -line NAND-CIRC program s.t.

Note that the “furthermore” part is extremely strong. It means that if the plaintext is even a little bit larger than the key, then we can already break the scheme in a very strong way. That is, there will be a pair of messages , (think of as “sell” and as “buy”) and an efficient strategy for Eve such that if Eve gets a ciphertext then she will be able to tell whether is an encryption of or with probability very close to . (We model breaking the scheme as Eve outputting or corresponding to whether the message sent was or . Note that we could have just as well modified Eve to output instead of and instead of . The key point is that a priori Eve only had a 50/50 chance of guessing whether Alice sent or but after seeing the ciphertext this chance increases to better than 99/1.) The condition can be relaxed to and even the weaker condition with essentially the same proof.

The proof follows along the lines of longkeysthm but this time paying attention to the computational aspects. If then for every plaintext and ciphertext , we can efficiently tell whether there exists such that . So, to prove this result we need to show that if the plaintexts are long enough, there would exist a pair such that the probability that a random encryption of also is a valid encryption of will be very small. The details of how to show this are below.

We focus on showing only the “furthermore” part since it is the more interesting and the other part follows by essentially the same proof.

Suppose that is such an encryption, let be large enough, and let . For every we define to be the set of all valid encryption of . That is . As in the proof of longkeysthm, since there are keys , for every .

We denote by the set . We define our algorithm to output on input if and to output otherwise. This can be implemented in polynomial time if , since the key can serve the role of an efficiently verifiable solution. (Can you see why?) Clearly and so in the case that gets an encryption of then she guesses correctly with probability . The remainder of the proof is devoted to showing that there exists such that , which will conclude the proof by showing that guesses wrongly with probability at most .

Consider now the following probabilistic experiment (which we define solely for the sake of analysis). We consider the sample space of choosing uniformly in and define the random variable to equal if and only if . For every , the map is one-to-one, which means that the probability that is equal to the probability that which is . So by the linearity of expectation .

We will now use the following extremely simple but useful fact known as the *averaging principle* (see also averagingprinciplerem): for every random variable , if , then with positive probability . (Indeed, if with probability one, then the expected value of will have to be larger than , just like you can’t have a class in which all students got A or A- and yet the overall average is B+.) In our case it means that with positive probability . In other words, there exists some such that . Yet this means that if we choose a random , then the probability that is at most . So, in particular if we have an algorithm that outputs if and outputs otherwise, then and which will be smaller than if .

In retrospect breakingcryptowithnp is perhaps not surprising. After all, as we’ve mentioned before it is known that the Optimal PRG conjecture (which is the basis for the derandomized one-time pad encryption) is *false* if (and in fact even if or even ).

## Public key cryptography

People have been dreaming about heavier-than-air flight since at least the days of Leonardo Da Vinci (not to mention Icarus from the greek mythology). Jules Verne wrote with rather insightful details about going to the moon in 1865. But, as far as I know, in all the thousands of years people have been using secret writing, until about 50 years ago no one has considered the possibility of communicating securely without first exchanging a shared secret key.

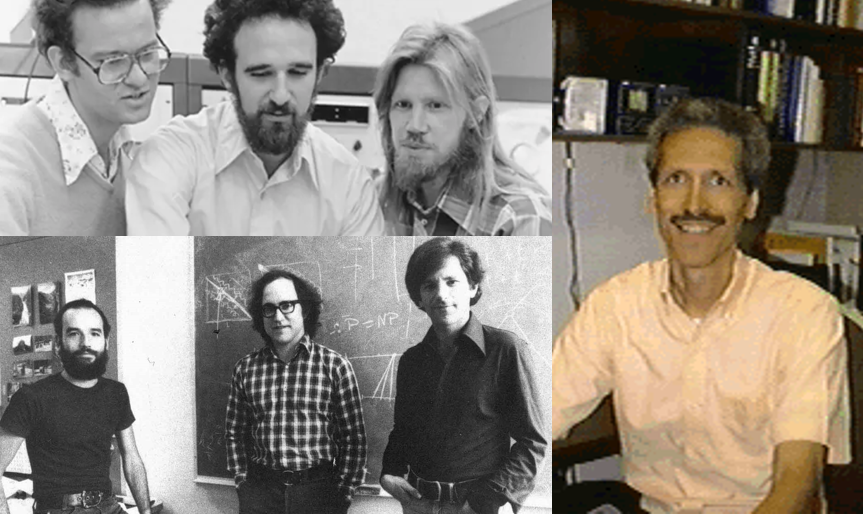
Yet in the late 1960’s and early 1970’s, several people started to question this “common wisdom”. Perhaps the most surprising of these visionaries was an undergraduate student at Berkeley named Ralph Merkle. In the fall of 1974 Merkle wrote in a [project proposal](http://www.merkle.com/1974/) for his computer security course that while “it might seem intuitively obvious that if two people have never had the opportunity to prearrange an encryption method, then they will be unable to communicate securely over an insecure channel… I believe it is false”. The project proposal was rejected by his professor as “not good enough”. Merkle later submitted a paper to the communication of the ACM where he apologized for the lack of references since he was unable to find any mention of the problem in the scientific literature, and the only source where he saw the problem even *raised* was in a science fiction story. The paper was rejected with the comment that “Experience shows that it is extremely dangerous to transmit key information in the clear.” Merkle showed that one can design a protocol where Alice and Bob can use invocations of a hash function to exchange a key, but an adversary (in the random oracle model, though he of course didn’t use this name) would need roughly invocations to break it. He conjectured that it may be possible to obtain such protocols where breaking is *exponentially harder* than using them, but could not think of any concrete way to doing so.

We only found out much later that in the late 1960’s, a few years before Merkle, James Ellis of the British Intelligence agency GCHQ was [having similar thoughts](http://cryptome.org/jya/ellisdoc.htm). His curiosity was spurred by an old World-War II manuscript from Bell labs that suggested the following way that two people could communicate securely over a phone line. Alice would inject noise to the line, Bob would relay his messages, and then Alice would subtract the noise to get the signal. The idea is that an adversary over the line sees only the sum of Alice’s and Bob’s signals, and doesn’t know what came from what. This got James Ellis thinking whether it would be possible to achieve something like that digitally. As Ellis later recollected, in 1970 he realized that in principle this should be possible, since he could think of an hypothetical black box that on input a “handle” and plaintext would give a “ciphertext” and that there would be a secret key corresponding to , such that feeding and to the box would recover . However, Ellis had no idea how to actually instantiate this box. He and others kept giving this question as a puzzle to bright new recruits until one of them, Clifford Cocks, came up in 1973 with a candidate solution loosely based on the factoring problem; in 1974 another GCHQ recruit, Malcolm Williamson, came up with a solution using modular exponentiation.

But among all those thinking of public key cryptography, probably the people who saw the furthest were two researchers at Stanford, Whit Diffie and Martin Hellman. They realized that with the advent of electronic communication, cryptography would find new applications beyond the military domain of spies and submarines, and they understood that in this new world of many users and point to point communication, cryptography will need to scale up. Diffie and Hellman envisioned an object which we now call “trapdoor permutation” though they called “one way trapdoor function” or sometimes simply “public key encryption”. Though they didn’t have full formal definitions, their idea was that this is an injective function that is easy (e.g., polynomial-time) to *compute* but hard (e.g., exponential-time) to *invert*. However, there is a certain *trapdoor*, knowledge of which would allow polynomial time inversion. Diffie and Hellman argued that using such a trapdoor function, it would be possible for Alice and Bob to communicate securely *without ever having exchanged a secret key*. But they didn’t stop there. They realized that protecting the *integrity* of communication is no less important than protecting its *secrecy*. Thus they imagined that Alice could “run encryption in reverse” in order to certify or *sign* messages.

At the point, Diffie and Hellman were in a position not unlike physicists who predicted that a certain particle should exist but without any experimental verification. Luckily they [met Ralph Merkle](http://cr.yp.to/bib/1988/diffie.pdf), and his ideas about a probabilistic *key exchange protocol*, together with a suggestion from their Stanford colleague [John Gill](http://hdl.handle.net/11299/107353), inspired them to come up with what today is known as the [Diffie Hellman Key Exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange) (which unbeknownst to them was found two years earlier at GCHQ by Malcolm Williamson). They published their paper [“New Directions in Cryptography”](https://www-ee.stanford.edu/~hellman/publications/24.pdf) in 1976, and it is considered to have brought about the birth of modern cryptography.

The Diffie-Hellman Key Exchange is still widely used today for secure communication. However, it still felt short of providing Diffie and Hellman’s elusive trapdoor function. This was done the next year by Rivest, Shamir and Adleman who came up with the RSA trapdoor function, which through the framework of Diffie and Hellman yielded not just encryption but also signatures. (A close variant of the RSA function was discovered earlier by Clifford Cocks at GCHQ, though as far as I can tell Cocks, Ellis and Williamson did not realize the application to digital signatures.) From this point on began a flurry of advances in cryptography which hasn’t died down till this day.

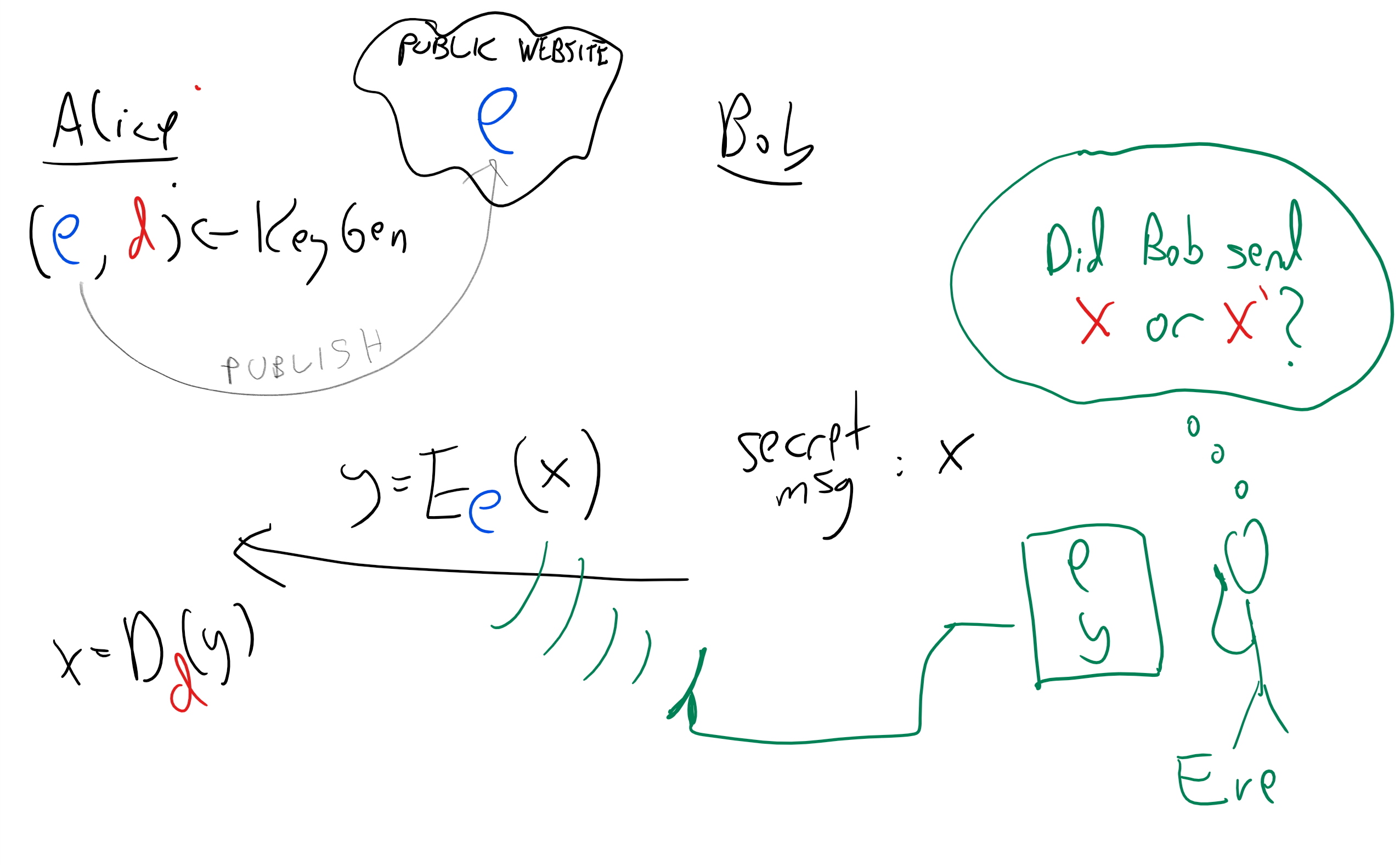


Top left: Ralph Merkle, Martin Hellman and Whit Diffie, who together came up in 1976 with the concept of *public key encryption* and a *key exchange protocol*. Bottom left: Adi Shamir, Ron Rivest, and Leonard Adleman who, following Diffie and Hellman’s paper, discovered the RSA function that can be used for public key encryption and digital signatures. Interestingly, one can see the equation on the blackboard behind them. Right: John Gill, who was the first person to suggest to Diffie and Hellman that they use modular exponentiation as an easy-to-compute but hard-to-invert function.

### Defining public key encryption

A *public key encryption* consists of a triple of algorithms:

* The *key generation algorithm*, which we denote by or for short, is a randomized algorithm that outputs a pair of strings where is known as the *public* (or *encryption*) key, and is known as the *private* (or *decryption*) key. The key generation algorithm gets as input (i.e., a string of ones of length ). We refer to as the *security parameter* of the scheme. The bigger we make , the more secure the encryption will be, but also the less efficient it will be.
* The *encryption algorithm*, which we denote by , takes the encryption key and a plaintext , and outputs the ciphertext .
* The *decryption algorithm*, which we denote by , takes the decryption key and a ciphertext , and outputs the plaintext .



In a *public key encryption*, Alice generates a private/public keypair , publishes and keeps secret. To encrypt a message for Alice, one only needs to know . To decrypt it we need to know .

We now make this a formal definition:

A *computationally secret public key encryption* with plaintext length is a triple of randomized polynomial-time algorithms that satisfy the following conditions:

* For every , if is output by with positive probability, and , then with probability one.
* For every polynomial , and sufficiently large , if is a NAND-CIRC program of at most lines then for every , , where this probability is taken over the coins of and .

publickeyencdef allows and to be *randomized* algorithms. In fact, it turns out that it is *necessary* for to be randomized to obtain computational secrecy. It also turns out that, unlike the private key case, we can transform a public-key encryption that works for messages that are *only one bit long* into a public-key encryption scheme that can encrypt arbitrarily long messages, and in particular messages that are *longer than the key*. In particular this means that we cannot obtain a perfectly secret public-key encryption scheme even for one-bit long messages (since it would imply a perfectly secret public-key, and hence in particular private-key, encryption with messages longer than the key).

We will not give full constructions for public key encryption schemes in this chapter, but will mention some of the ideas that underlie the most widely used schemes today. These generally belong to one of two families:

* *Group theoretic constructions* based on problems such as *integer factoring* and the *discrete logarithm* over finite fields or elliptic curves.
* *Lattice/coding based constructions* based on problems such as the *closest vector in a lattice* or *bounded distance decoding*.

Group-theory based encryptions such as the RSA cryptosystem, the Diffie-Hellman protocol, and Elliptic-Curve Cryptography, are currently more widely implemented. But the lattice/coding schemes are recently on the rise, particularly because the known group theoretic encryption schemes can be broken by *quantum computers*, which we discuss in quantumchap.

### Diffie-Hellman key exchange

As just one example of how public key encryption schemes are constructed, let us now describe the Diffie-Hellman key exchange. We describe the Diffie-Hellman protocol in a somewhat of an informal level, without presenting a full security analysis.

The computational problem underlying the Diffie Hellman protocol is the *discrete logarithm problem*. Let’s suppose that is some integer. We can compute the map and also its *inverse* . (For example, we can compute a logarithm is by *binary search*: start with some interval that is guaranteed to contain . We can then test whether the interval’s midpoint satisfies , and based on that halve the size of the interval.)

However, suppose now that we use *modular arithmetic* and work modulo some prime number . If has binary digits and is in then we can compute the map in time polynomial in . (This is not trivial, and is a great exercise for you to work this out; as a hint, start by showing that one can compute the map using modular multiplications modulo , if you’re stumped, you can look up [this Wikipedia entry](https://en.wikipedia.org/wiki/Exponentiation_by_squaring).) On the other hand, because of the “wraparound” property of modular arithmetic, we cannot run binary search to find the inverse of this map (known as the *discrete logarithm*). In fact, there is no known polynomial-time algorithm for computing this discrete logarithm map map , where we define as the number such that .

The Diffie-Hellman protocol for Bob to send a message to Alice is as follows:

* **Alice:** Chooses to be a random bit long prime (which can be done by choosing random numbers and running a primality testing algorithm on them), and and at random in . She sends to Bob the triple .
* **Bob:** Given the triple , Bob sends a message to Alice by choosing at random in , and sending to Alice the pair where is some “representation function” that maps to . (The function does not need to be one-to-one and you can think of as simply outputting of the bits of in the natural binary representation, it does need to satisfy certain technical conditions which we omit in this description.)
* **Alice:** Given , Alice recovers by outputting .

The correctness of the protocol follows from the simple fact that for every and this still holds if we work modulo . Its security relies on the computational assumption that computing this map is hard, even in a certain “average case” sense (this computational assumption is known as the [Decisional Diffie Hellman assumption](https://en.wikipedia.org/wiki/Decisional_Diffie%E2%80%93Hellman_assumption)). The Diffie-Hellman key exchange protocol can be thought of as a public key encryption where the Alice’s first message is the public key, and Bob’s message is the encryption.

One can think of the Diffie-Hellman protocol as being based on a “trapdoor pseudorandom generator” whereas the triple looks “random” to someone that doesn’t know , but someone that does know can see that raising the second element to the -th power yields the third element. The Diffie-Hellman protocol can be described abstractly in the context of any [finite Abelian group](https://en.wikipedia.org/wiki/Abelian_group) for which we can efficiently compute the group operation. It has been implemented on other groups than numbers modulo , and in particular [Elliptic Curve Cryptography (ECC)](https://en.wikipedia.org/wiki/Elliptic-curve_cryptography) is obtained by basing the Diffie Hellman on elliptic curve groups which gives some practical advantages. Another common group theoretic basis for key-exchange/public key encryption protocol is the RSA function. A big disadvantage of Diffie-Hellman (both the modular arithmetic and elliptic curve variants) and RSA is that both schemes can be broken in polynomial time by a *quantum computer*. We will discuss quantum computing later in this course.

## Other security notions

There is a great deal to cryptography beyond just encryption schemes, and beyond the notion of a passive adversary. A central objective is *integrity* or *authentication*: protecting communications from being modified by an adversary. Integrity is often more fundamental than secrecy: whether it is a software update or viewing the news, you might often not care about the communication being secret as much as that it indeed came from its claimed source. *Digital signature schemes* are the analog of public key encryption for authentication, and are widely used (in particular as the basis for [public key certificates](https://en.wikipedia.org/wiki/Public_key_certificate)) to provide a foundation of trust in the digital world.

Similarly, even for encryption, we often need to ensure security against *active attacks*, and so notions such as non-malleability and [adaptive chosen ciphertext](https://en.wikipedia.org/wiki/Adaptive_chosen-ciphertext_attack) security have been proposed. An encryption scheme is only as secure as the secret key, and mechanisms to make sure the key is generated properly, and is protected against refresh or even compromise (i.e., [forward secrecy](https://en.wikipedia.org/wiki/Forward_secrecy)) have been studied as well. Hopefully this chapter provides you with some appreciation for cryptography as an intellectual field, but does not imbue you with a false self of confidence in implementing it.

*Cryptographic hash functions* is another widely used tool with a variety of uses, including extracting randomness from high entropy sources, achieving hard-to-forge short “digests” of files, protecting passwords, and much more.

## Magic

Beyond encryption and signature schemes, cryptographers have managed to obtain objects that truly seem paradoxical and “magical”. We briefly discuss some of these objects. We do not give any details, but hopefully this will spark your curiosity to find out more.

### Zero knowledge proofs

On October 31, 1903, the mathematician Frank Nelson Cole, gave an hourlong lecture to a meeting of the American Mathematical Society where he did not speak a single word. Rather, he calculated on the board the value which is equal to , and then showed that this number is equal to . Cole’s proof showed that is not a prime, but it also revealed additional information, namely its actual factors. This is often the case with proofs: they teach us more than just the validity of the statements.

In *Zero Knowledge Proofs* we try to achieve the opposite effect. We want a proof for a statement where we can *rigorously show* that the proofs reveals *absolutely no additional information about*  beyond the fact that it is true. This turns out to be an extremely useful object for a variety of tasks including authentication, secure protocols, voting, [anonymity in cryptocurrencies](https://z.cash/technology/zksnarks.html), and more. Constructing these objects relies on the theory of completeness. Thus this theory that originally was designed to give a *negative result* (show that some problems are hard) ended up yielding *positive applications*, enabling us to achieve tasks that were not possible otherwise.

### Fully homomorphic encryption

Suppose that we are given a bit-by-bit encryption of a string . By design, these ciphertexts are supposed to be “completely unscrutable” and we should not be able to extract any information about ’s from it. However, already in 1978, Rivest, Adleman and Dertouzos observed that this does not imply that we could not *manipulate* these encryptions. For example, it turns out the security of an encryption scheme does not immediately rule out the ability to take a pair of encryptions and and compute from them *without knowing the secret key* . But do there exist encryption schemes that allow such manipulations? And if so, is this a bug or a feature?

Rivest et al already showed that such encryption schemes could be *immensely* useful, and their utility has only grown in the age of cloud computing. After all, if we can compute NAND then we can use this to run any algorithm on the encrypted data, and map to . For example, a client could store their secret data in encrypted form on the cloud, and have the cloud provider perform all sorts of computation on these data without ever revealing to the provider the private key, and so without the provider *ever learning any information* about the secret data.

The question of *existence* of such a scheme took much longer time to resolve. Only in 2009 Craig Gentry gave the first construction of an encryption scheme that allows to compute a universal basis of gates on the data (known as a *Fully Homomorphic Encryption scheme* in crypto parlance). Gentry’s scheme left much to be desired in terms of efficiency, and improving upon it has been the focus of an intensive research program that has already seen significant improvements.

### Multiparty secure computation

Cryptography is about enabling mutually distrusting parties to achieve a common goal. Perhaps the most general primitive achieving this objective is [secure multiparty computation](https://en.wikipedia.org/wiki/Secure_multi-party_computation). The idea in secure multiparty computation is that parties interact together to compute some function where is the private input of the -th party. The crucial point is that there is *no commonly trusted party or authority* and that nothing is revealed about the secret data beyond the function’s output. One example is an *electronic voting protocol* where only the total vote count is revealed, with the privacy of the individual voters protected, but without having to trust any authority to either count the votes correctly or to keep information confidential. Another example is implementing a [second price (aka Vickrey) auction](https://en.wikipedia.org/wiki/Vickrey_auction) where parties submit bids to an item owned by the -th party, and the item goes to the highest bidder but at the price of the *second highest bid*. Using secure multiparty computation we can implement second price auction in a way that will ensure the secrecy of the numerical values of all bids (including even the top one) except the second highest one, and the secrecy of the identity of all bidders (including even the second highest bidder) except the top one. We emphasize that such a protocol requires no trust even in the auctioneer itself, that will also not learn any additional information. Secure multiparty computation can be used even for computing *randomized* processes, with one example being playing Poker over the net without having to trust any server for correct shuffling of cards or not revealing the information.

* We can formally define the notion of security of an encryption scheme.
* *Perfect secrecy* ensures that an adversary does not learn *anything* about the plaintext from the ciphertext, regardless of their computational powers.
* The one-time pad is a perfectly secret encryption with the length of the key equaling the length of the message. No perfectly secret encryption can have key shorter than the message.
* *Computational secrecy* can be as good as perfect secrecy since it ensures that the advantage that computationally bounded adversaries gain from observing the ciphertext is exponentially small. If the optimal PRG conjecture is true then there exists a computationally secret encryption scheme with messages that can be (almost) *exponentially bigger* than the key.
* There are many cryptographic tools that go well beyond private key encryption. These include *public key encryption*, *digital signatures* and *hash functions*, as well as more “magical” tools such as *multiparty secure computation*, *fully homomorphic encryption*, *zero knowledge proofs*, and many others.

## Exercises

## Bibliographical notes

Much of this text is taken from [my lecture notes on cryptography](https://intensecrypto.org).

Shannon’s manuscript was written in 1945 but was classified, and a partial version was only published in 1949. Still it has revolutionized cryptography, and is the forerunner to much of what followed.

The Venona project’s history is described in [this document](http://nsarchive.gwu.edu/NSAEBB/NSAEBB278/01.PDF). Aside from Grabeel and Zubko, credit to the discovery that the Soviets were reusing keys is shared by Lt. Richard Hallock, Carrie Berry, Frank Lewis, and Lt. Karl Elmquist, and there are others that have made important contribution to this project. See pages 27 and 28 in the document.

In a [1955 letter to the NSA](https://www.nsa.gov/news-features/declassified-documents/nash-letters/assets/files/nash_letters1.pdf) that only recently came forward, John Nash proposed an “unbreakable” encryption scheme. He wrote *“I hope my handwriting, etc. do not give the impression I am just a crank or circle-squarer…. The significance of this conjecture [that certain encryption schemes are exponentially secure against key recovery attacks] .. is that it is quite feasible to design ciphers that are effectively unbreakable.”*. John Nash made seminal contributions in mathematics and game theory, and was awarded both the Abel Prize in mathematics and the Nobel Memorial Prize in Economic Sciences. However, he has struggled with mental illness throughout his life. His biography, [A Beautiful Mind](https://en.wikipedia.org/wiki/A_Beautiful_Mind_(book)) was made into a popular movie. It is natural to compare Nash’s 1955 letter to the NSA to Gödel’s letter to von Neumann we mentioned before. From the theoretical computer science point of view, the crucial difference is that while Nash informally talks about exponential vs polynomial computation time, he does not mention the word “Turing Machine” or other models of computation, and it is not clear if he is aware or not that his conjecture can be made mathematically precise (assuming a formalization of “sufficiently complex types of enciphering”).

The definition of computational secrecy we use is the notion of *computational indistinguishability* (known to be equivalent to *semantic security*) that was given by Goldwasser and Micali in 1982.

Although they used a different terminology, Diffie and Hellman already made clear in their paper that their protocol can be used as a public key encryption, with the first message being put in a “public file”. In 1985, ElGamal showed how to obtain a *signature scheme* based on the Diffie Hellman ideas, and since he described the Diffie-Hellman encryption scheme in the same paper, the public key encryption scheme originally proposed by Diffie and Hellman is sometimes also known as ElGamal encryption.

[My survey](https://eccc.weizmann.ac.il/report/2017/065/) contains a discussion on the different types of public key assumptions. While the standard elliptic curve cryptographic schemes are as susceptible to quantum computers as Diffie-Hellman and RSA, their main advantage is that the best known classical algorithms for computing discrete logarithms over elliptic curve groups take time for some where is the number of bits to describe a group element. In contrast, for the multiplicative group modulo a prime the best algorithm take time which means that (assuming the known algorithms are optimal) we need to set the prime to be bigger (and so have larger key sizes with corresponding overhead in communication and computation) to get the same level of security.

Zero-knowledge proofs were constructed by Goldwasser, Micali, and Rackoff in 1982, and their wide applicability was shown (using the theory of completeness) by Goldreich, Micali, and Wigderson in 1986.

Two party and multiparty secure computation protocols were constructed (respectively) by Yao in 1982 and Goldreich, Micali, and Wigderson in 1987. The latter work gave a general transformation from security against passive adversaries to security against active adversaries using zero knowledge proofs.

1. Here is a nice exercise: compute (up to an order of magnitude) the probability that a 50-letter long message composed of random letters will end up not containing the letter “L”. [↑](#footnote-ref-35)
2. The actual quote is “Il faut qu’il n’exige pas le secret, et qu’il puisse sans inconvénient tomber entre les mains de l’ennemi” loosely translated as “The system must not require secrecy and can be stolen by the enemy without causing trouble”. According to Steve Bellovin the NSA version is “assume that the first copy of any device we make is shipped to the Kremlin”. [↑](#footnote-ref-44)