Skeleton for LaTeX articles

Óscar Pereira[1]

18th January 2021

[1]{https://, oscar@}randomwalk.eu.

Contents

1 Introduction

> *Cryptography is not the solution to your security problems. It might be part of the solution, or it might be part of the problem.*
>
> N. Ferguson and B. Schneier

1.1 On Problems and Solutions

The attempt to achieve "secret writing" is probably as old as writing itself. And until the early 1970s and early 1980s, it was mostly an art form: one used intuition and *ad hoc* reasoning, and little else. That would change radically from that point onwards: it became more scientific (and certainly more rigorous), although it still requires *«a healthy dose of the black magic that we call experience»* (Ferguson and Schneier [5], p. 7).

Additionally, in the past the techniques for this kind of "secret writing", were themselves a closely guarded secret. While perhaps understandable, this led to an abundance of what is usually dubbed "snake oil": systems that might look like they were secure, while in reality being anything but. Eventually, the black magic of experience (in what regards to this particular point) was distilled into what is known as **Kerckhoff's principle**: when designing a cryptographic system, one must assume that all the details of the system are known to the cryptanalysts—all except the key(s), that is. This of course, does not mean that we should **advertise** all the details of the system to the wider world; only that when analysing security, we must not rely on the secrecy of anything, except the key.[1]

However, at least in what concerns the development of cryptographic primitives and protocols, that is exactly what happens: all the details are published, which acts as an invitation to everyone else to try to attack that construction (it's a fun field…). If no practical attacks are found after an acceptable period of analysis (usually years), and if it is fast enough for the intended purpose, then it becomes a candidate for standardisation—which is a sort of first step to widespread adoption.

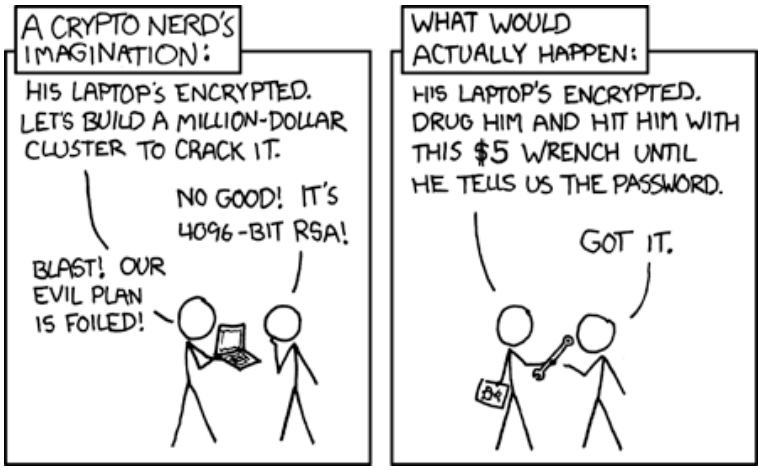
One of the results of this way of doing things, is that cryptography has become a huge field. The main goal of this course is precisely to help the student navigate and find his way amidst the myriad of different constructions—and acronyms!—that he will no doubt encounter in his future (technological) endeavours. And, equally important, to understand what cryptography *cannot* provide. As Ferguson and Schneier explain:

The epigraph comes from Ferguson and Schneier's *Practical Cryptography* [5], §2.6.

[1] Steve Bellovin notes that **the NSA itself** employs a somewhat similar mindset: *«A former official at NSA's National Computer Security Center told me that the standard assumption there was that serial number 1 of any new device was delivered to the Kremlin.»* [3].

> Too many engineers consider cryptography to be a sort of magic security dust that they can sprinkle over their hardware or software, and which will imbue those products with the mythical property of "security".[2]

By the end of the term, the student will hopefully be able to see why cryptography is just one component of a much larger and complex system. A system that ultimately, must include good old cunning human beings as well:



1.2 On Models, Definitions, and Paranoia

Our definitions of security usually take the form of defining a task that a notional adversary, that is trying to attack our system, must *not* be able to accomplish. Let a and b be two tasks, and assume that a is *easier* than b. Then it should be straightforward to see that (informally speaking):

> Being able to do the hard task (b) implies being able to do the easy task (a).

And conversely:

> Not being able to do the easy task (a) implies not being able to do the hard task (b).

Now think of security definitions. Intuitively, we would want to say that if a given construction verifies a strong notion of security, then it also verifies a weak(er) notion of security (although the converse will usually be false). Taking this into account, together with the two implications above, means the following: **the easier the task the adversary is assumed *not* to be able to do, the stronger the corresponding security definition will be**.

In particular, the more said task borders on the trivial (or the more resources and computational power the adversary is assumed to have), the more the corresponding security definition approaches *unconditional* security—in comparison to which all other security definitions will be weaker.

Primitives vs. protocols. When designing a **cryptographic primitive**, one tries to design it to be as secure as possible—in other words, the stronger the security definition that the primitive verifies, the better. However, when designing a **protocol** (say TLS), that uses some cryptographic primitive (say a block cipher), the protocol designer should be conservative, and assume that

[2] See [5], p. xviii.

the primitive he will be using is as *weak* as possible. In other words, he wants to be able to say: TLS is so well designed, that it remains secure even when the underlying block cipher sucks! Of course this is not true for TLS—but I hope you get the point.

The above discussion is very informal (probably to a fault…), but the reader is advised to keep it in mind when reading the plethora of definitions introduced hereinafter—and to come back and re-read this section whenever necessary.

The paranoia model. Another thing the reader would do well to keep in mind is the following: often when reading descriptions of attacks, it is tempting to think: "but this would not be a problem in practice!", or "but this would never happen!". However, history—even recent history—has shown this to be a dangerous mindset indeed. Take the padding oracle attack (§3.6): it involves sending a few modified encrypted packets during a TLS session. In most circumstances, this would make the attack irrelevant, because when a TLS peer receives the *first* invalid packet, it shuts down the connection. However, it turned out that, when TLS is used with IMAP (an email retrieval protocol), there was a setting where the attack could work—and it allowed for **the user's password** to be recovered! This is why cryptographers are paranoid. If there is a theoretical chance something might fail, that is enough justification to go improve it.

1.3 On Keys—Secret, "Big", and Independent

Kerckhoff's principle states that all the security resides in the key, so it must be kept **secret**. There is also the problem of which **size** of key should one choose. This varies so wildly with so many things that a single, one-size-fits-all (no pun intended) answer, is impossible. The following should, however, be kept in mind:

- Larger keys are **necessary** for improved security, but are **not sufficient**.
- The larger the key, the slower the construction will be.
- Key sizes cannot be compared across different constructions; particularly between symmetric and asymmetric constructions. By way of example, today it is considered prudent security to use a 256 bit key for AES, but for RSA the minimum recommended key length is 2048!!

And one last warning: when mixing different cryptographic constructions in the same scheme, **always use independent keys!!!** Things can go badly wrong otherwise—see the end of §4.5.2 in [9] for an illuminating example.

2 | Symmetric Ciphers

The fundamental task of cryptography is encryption.

S. ARORA AND B. BARAK

2.1 Perfect Secrecy

Believe it or not, there is such a thing as perfectly secret symmetric cipher. In the following sense: the cryptogram c of message m under key k yields *no information* whatsoever about m. But what does it mean for c to yield "no information" about m? Well, consider the space of all possible plaintext messages $\mathcal{M}$. The adversary, who is trying to discover something about m, by analysing c, might know nothing at all about the probabilities of the different messages in $\mathcal{M}$, but more often than not, he will know something about them—for example, he might know that m is a message written in English, rather than just random data. Saying that c yields no information about m means that after seeing c, the probability distribution of $\mathcal{M}$ that he is able to compute is exactly the same as the one he could compute *without* having seen c. In other words, seeing the cryptogram does not help at all to narrow down the scope of possible plaintext messages.

More formally, consider an encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, with message space $\mathcal{M}$, ciphertext space $\mathcal{C}$, and key space $\mathcal{K}$. Let M be a random variable that corresponds to the plaintext message, and C a random variable that corresponds to the ciphertext obtained. We have the following definition:

Definition 2.1. *An encryption scheme* Π *is* perfectly secret *if for every probability distribution over its message space* $\mathcal{M}$*, for every message* m *in its message space, and for every cryptogram* c *on its ciphertext space such that* $Prob[C = c] > 0$*, the following holds:*[1]

$$\text{Prob}[M = m \mid C = c] = \text{Prob}[M = m] \tag{2.1}$$

Having the probability of c greater than 0 is a technicality to avoid conditioning over an event with 0 probability.

Using Bayes' Theorem, we can give an alternative definition of perfect secrecy:

Theorem 2.2. *An encryption scheme* Π *is* perfectly secret *if for every probability distribution over its message space* $\mathcal{M}$*, for every* $m, m' \in \mathcal{M}$*, and every* $c \in \mathcal{C}$*, the following holds:*[2]

$$\text{Prob}[\mathsf{Enc}_K(m) = c] = \text{Prob}[\mathsf{Enc}_K(m') = c] \tag{2.2}$$

The epigraph comes from Arora and Barak's *Computational Complexity* [1], §9.

[1] This corresponds to Definition 2.3 given in ([9], p. 29.)

[2] This corresponds to Lemma 2.4 given in ([9], p. 30.)

Informally this means that regardless of the plaintext message we encipher, we always obtain the same distribution for the ciphertext space. To give a more concrete example, if encrypting message m gave the uniform distribution on $\mathcal{C}$, then that would mean that when encrypting *any other message*, all the ciphertexts in $\mathcal{C}$ would be equi-probable.

The Vernam cipher (OTP), and the limitations of perfect secrecy. There is actually a dead simple construction that yields perfect secrecy: the **One-Time Pad**, that Vernam thought up circa 1917. For simplicity, assume that both $\mathcal{M}$ and $\mathcal{C}$ consist of the 26 letters `A`–`Z`. Enciphering is as simple as, for each letter of the plaintext, choose another letter randomly, add the two modulo 26, and the result is the corresponding letter of the ciphertext. Repeat the process until there are no more letters in the message. Those randomly chosen letters are the secret key. To decrypt, just subtract (mod 26) the key from the ciphertext.

Sounds simple enough, but the problem is that the key must be generated from a "true" random source—meaning, basically, that pseudorandom generators (§2.4) *cannot* be used. This also implies that the length of the key must be at least that of the plaintext message—or equivalently, the key space must be at least as large as the message space: $|\mathcal{K}| \geq |\mathcal{M}|$. This, it turns out, is not a limitation of the OTP, but rather a necessary—though **not sufficient**—condition for perfect secrecy.

To see why, consider an encryption scheme where $|\mathcal{K}| < |\mathcal{M}|$, and the following attack strategy: given a ciphertext c, we just decrypt c using every possible key. As the cardinality of the key space is less than that of the plaintext message space, the result will be a *subset* of $\mathcal{M}$—denote it $\mathcal{S}$. This immediately allows the following modification on the probability distribution of $\mathcal{M}$: all the messages in $\mathcal{M} \setminus \mathcal{S}$ can now be assigned probability 0. For the only messages for which there is a key that encrypts them to c are those in $\mathcal{S}$. And so having $|\mathcal{K}| < |\mathcal{M}|$ means that perfect secrecy is not attainable.

2.2 Security Definitions

We begin with the simplest definition: security against a passive attacker (eavesdropper), who sees only one ciphertext (for a given key). This is called **EAV-security**, and is defined by the following security game (denoted $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi}$):

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Adversary $\mathcal{A}(1^n)$ chooses two messages of equal length, m_0 and m_1.
3. We choose a uniform random bit $b \leftarrow \{0,1\}$, and then encrypt m_b, obtaining ciphertext $c \leftarrow Enc_k(m_b)$. c is given to $\mathcal{A}$. (c is called the *challenge ciphertext*.)
4. $\mathcal{A}$ outputs a bit b'. This is $\mathcal{A}$'s guess of the value of bit b.
5. If $\mathcal{A}$ guessed correctly (i.e. if $b' = b$), we say it *succeeded*. In that case, $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi} = 1$; otherwise $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi} = 0$.

The intuition is that the better the cipher, the harder it will be for the adversary to extract any information from the ciphertext, *regardless of what the corresponding plaintext message happened to be*. This is why we allow for the adversary itself to choose the plaintext messages. In the above game, if the adversary simply *guesses* a value for b', he will be correct with probability $1/2$. This happens if the cipher has perfect security; for concrete security, we are happy if the adversary is only able to do negligibly better than this.

Definition 2.3. *A private key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **perfectly indistinguishable** *if for all adversaries* $\mathcal{A}$*,* **including non-efficient ones***, it holds that:*[3]

$$\mathrm{Prob}\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi}(n) = 1\right] = \frac{1}{2} \tag{2.3}$$

The probability is taken over the randomness used by $\mathcal{A}$*, together with whatever randomness is used in the experiment (*Π *plus choosing the bit* b*).*

This is equivalent to perfect secrecy ([9], Lemma 2.6, p. 31). In practice, however, we give up perfect secrecy and non-efficient adversaries. We focus instead in *concrete security*, meaning constructions are indexed by a security parameter n, and efficient adversaries, meaning adversaries that run in *probabilistic polynomial time* (PPT). "Probabilistic" means the algorithm can "toss coins", i.e. make random choices.

The following definition is the concrete security equivalent of perfect secrecy:

Definition 2.4. *A private key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **EAV-secure** *if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that, for all* n*:*[4]

$$\mathrm{Prob}\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n) \tag{2.4}$$

The probability is taken over the randomness used by $\mathcal{A}$*, together with whatever randomness is used in the experiment (*Π *plus choosing the bit* b*).*

The next step is to consider what happens if the adversary sees more than one ciphertext (for a given key). Think of a *deterministic* cipher, i.e. one where once fixed a key, encrypting the same plaintext always produces the same ciphertext (e.g. the OTP): if the adversary is observing the encrypted traffic, and all of a sudden sees a ciphertext that he has seen already, then he will know that there was a message that was sent twice. This violates the above intuition of the adversary not being able to extract any information from all the ciphertext he sees, and hence this scenario requires a stronger security definition.

That stronger notion is **MULT-security**, which corresponds to the following security game (denoted $\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\,\Pi}$):

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Adversary $\mathcal{A}(1^n)$ chooses two *vectors* of messages, $\boldsymbol{M_0} = (m_{0,1} \ldots, m_{1,t})$ and $\boldsymbol{M_1} = (m_{1,1}, \ldots, m_{1,t})$, having $\left|m_{0,i}\right| = \left|m_{1,i}\right|$, for all $1 \leq i \leq t$.
3. We choose a uniform random bit $b \leftarrow \{0, 1\}$, and then encrypt $\boldsymbol{M}_b$, obtaining ciphertext vector $C = (c_1, \ldots, c_t)$, where $c_i \leftarrow Enc_k(m_{b,i})$. C is given to $\mathcal{A}$.
4. $\mathcal{A}$ outputs a bit b'. This is $\mathcal{A}$'s guess of the value of bit b.
5. If $\mathcal{A}$ guessed correctly (i.e. if $b' = b$), we say it *succeeded*. In that case, $\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\,\Pi} = 1$; otherwise $\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\,\Pi} = 0$.

The security condition is similar to definition 2.4, *mutatis mutandis*:

[3] This corresponds to Definition 2.5 given in ([9], p. 31).
[4] This corresponds to Definition 3.8 given in ([9], p. 55).

Definition 2.5. *A private key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **MULT-secure** *if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that, for all* n*:*[5]

$$\mathrm{Prob}\left[\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n) \tag{2.5}$$

The probability is taken over the randomness used by $\mathcal{A}$*, together with whatever randomness is used in the experiment (*Π *plus choosing the bit* b*).*

Note that MULT-security includes, as a special case, EAV-security: indeed one just has to use vectors of size one (i.e. set $t = 1$).[6] But even this more strict definition of security for encryption schemes (confidentiality), does not suffice for all situations. Indeed, the adversary can come into possession of pair(s) (plaintext, ciphertext)—in which case it becomes trivial to win the EAV-security game with probability 1 (why?). This is a known-plaintext attack, and we can consider this as a special case of the scenario where the adversary can *choose* the plaintext messages that get encrypted (and which ciphertext he subsequently obtains; this is a chosen plaintext attack).[7]

Modelling a known plaintext attack simply means the adversary has access to some fixed information (the (plaintext, ciphertext) pairs). The fact that he can choose what plaintexts he wants encrypted is modelled giving him access to an *encryption oracle* $\mathsf{Enc}_k(\cdot)$. Intuitively, the generalisation of MULT-security means that we now again deal with lists of messages, but to aid him in his task, the adversary can now request the encryption of plaintexts of his choice. In fact, we will now make the adversary even stronger, by allowing him to *adapt* his future queries, based on the result of past queries. That is, instead of the adversary only being allowed to give a list of plaintexts, and request the corresponding ciphertexts, he can now ask for the encryption of a given plaintext, and analyse that ciphertext, before deciding what plaintext he will ask to be encrypted next.

Such a capability is modelled with a **left-right oracle**, denoted $\mathsf{LR}_{k,b}$, that given two messages m_0 and m_1, outputs $\mathsf{Enc}_k(m_b)$. Thus we emulate the MULT-security experiment with the queries $\mathsf{LR}_{k,b}(m_{0,1}, m_{1,1}), \ldots, \mathsf{LR}_{k,b}(m_{0,t}, m_{1,t})$, which yield the ciphertexts $\mathsf{Enc}_k(m_{b,i})$, for $1 \leq i \leq t$. (But note that after receiving, say, $\mathsf{Enc}_k(m_{b,1})$, the adversary can decide on the next message that gets encrypted—i.e. it is adaptative.) We now have the following security experiment (denoted $\mathsf{PrivK}^{\mathsf{LR\text{-}cpa}}_{\mathcal{A},\,\Pi}$):

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Choose a bit $b \leftarrow \{0, 1\}$.
3. Give the adversary $\mathcal{A}(1^n)$ access to oracle $\mathsf{LR}_{k,b}(\cdot, \cdot)$, defined as above.
4. $\mathcal{A}$ outputs bit b'.
5. If $b' = b$, we say that A has *succeeded*. In that case, $\mathsf{PrivK}^{\mathsf{LR\text{-}cpa}}_{\mathcal{A},\,\Pi} = 1$. Otherwise $\mathsf{PrivK}^{\mathsf{LR\text{-}cpa}}_{\mathcal{A},\,\Pi} = 0$.

And the by now customary security definition:

Definition 2.6. *A private key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **CPA-secure for multiple encryptions** *if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that, for all* n*:*[8]

$$\mathrm{Prob}\left[\mathsf{PrivK}^{\mathsf{LR\text{-}cpa}}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n) \tag{2.6}$$

[5] This corresponds to Definition 3.19 given in ([9], p. 71).

[6] This also entails that if a scheme is not EAV-secure, it cannot possibly be MULT-secure.

[7] Both of these are realistic scenarios; see Katz and Lindell [9], esp. chaps. 1–3.

[8] This corresponds to Definition 3.23 given in ([9], p. 76).

The probability is taken over the randomness used by $\mathcal{A}$, together with whatever randomness is used in the experiment (Π plus choosing the bit b).

We say CPA-secure *for multiple encryptions* because the adversary can, in the game above, obtain an arbitrary number of ciphertexts. But we can also consider a simplified scenario, of CPA-security for a single encryption. This is what we do in the following experiment (denoted $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\,\Pi}$):

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Give adversary $\mathcal{A}(1^n)$ access to the encryption oracle $\mathsf{Enc}_k(\cdot)$.[9] $\mathcal{A}$ outputs m_0 and m_1 of the same length.
3. Choose a uniform bit $b \leftarrow \{0,1\}$, and then compute $c \leftarrow \mathsf{Enc}_k(m_b)$. Give c to $\mathcal{A}$.
4. $\mathcal{A}$ continues to have access to $\mathsf{Enc}_k(\cdot)$, and outputs a bit b'.
5. If $b' = b$, we say that A has *succeeded*. In that case, $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\,\Pi} = 1$. Otherwise $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\,\Pi} = 0$.

Note that here the adversary only sees one ciphertext. It is straightforward to modify the above definition to this new scenario:

Definition 2.7. *A private key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is* **CPA-secure for a single encryption** *if for all PPT adversaries $\mathcal{A}$, there is a negligible function* negl *such that, for all n:*[10]

$$\mathrm{Prob}\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n) \tag{2.7}$$

The probability is taken over the randomness used by $\mathcal{A}$, together with whatever randomness is used in the experiment (Π plus choosing the bit b).

If a given encryption scheme if CPA-secure for multiple encryptions, then it is clear that it is also CPA-secure for a single encryption. But surprisingly, **the converse also holds!** The following is a direct quote of Theorem 3.24 ([9], p. 76):

Theorem 2.8. *Any private-key encryption scheme that is CPA-secure is also CPA-secure for* **multiple** *encryptions.*[11]

This is why we can speak just of CPA-security, without any ambiguity. This also simplifies proofs: if one can prove a scheme CPA-secure for a single encryption, then CPA-security for multiple encryptions comes "for free"—i.e. without additional work.

Chosen Ciphertext Security (CCA-security). The final capability to handed to the adversary, is to allow him to *decrypt* ciphertexts of his choice. With one obvious exception: he is not allowed to request the decryption of the challenge ciphertext—otherwise he could always win the security game with probability 1. Now the reader may wonder: is this realistic? After all, it is hard to imagine an honest party happily deciphering ciphertexts at the adversary's behest. Nevertheless, such things do happen, and we will see an example of this with the padding oracle attack (§3.6).

The CCA security experiment: $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\,\Pi}$).

[9] Note that this is **not** the left-right oracle of the previous game, but a simpler one that takes one plaintext and returns its corresponding encryption under key k.

[10] This corresponds to Definition 3.22 given in ([9], p. 75).

[11] This corresponds to Theorem 3.24 given in ([9], p. 76). See therein for a reference to a proof.

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Give adversary $\mathcal{A}(1^n)$ access to the encryption oracle $\mathsf{Enc}_k(\cdot)$ and a decryption oracle $\mathsf{Dec}_k(\cdot)$. $\mathcal{A}$ outputs m_0 and m_1 of the same length.
3. Choose a uniform bit $b \leftarrow \{0,1\}$, and then compute $c \leftarrow \mathsf{Enc}_k(m_b)$. Give c to $\mathcal{A}$.
4. $\mathcal{A}$ continues to have access to $\mathsf{Enc}_k(\cdot)$ and $\mathsf{Dec}_k(\cdot)$, but is not allowed to query the latter on c. Eventually $\mathcal{A}$ outputs a bit b'.
5. If $b' = b$, we say that A has *succeeded*. In that case, $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\,\Pi} = 1$. Otherwise $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\,\Pi} = 0$.

Note that here the adversary only sees one ciphertext. It is straightforward to modify the above definition to this new scenario:

Definition 2.9. *A private key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **CCA-secure** *if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that, for all* n*:*[12]

$$\mathrm{Prob}\left[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n) \tag{2.8}$$

The probability is taken over the randomness used by $\mathcal{A}$*, together with whatever randomness is used in the experiment (*Π *plus choosing the bit* b*).*

There exists an analog of theorem 2.8 for CCA-security, namely that CCA-security for a single encryption implies CCA-security for multiple encryptions. Which is why the definition only speaks of "CCA-security".

Constructing schemes that verify this very strong notion of security requires a primitive that is the topic of chapter 3: message authentication codes. Hence, further discussion takes place therein.

2.3 Block Ciphers

Symmetric ciphers can process their input (the plaintext message to enciphered) either bit by bit, or blocks of bits at a time. The former case requires PRNGs (§2.4), which is why it is discussed afterwords—and in this section, we consider only the latter case.

In practice most block ciphers map some number n of bits to the same number of bits, which can be modelled as a *pseudorandom permutation* (PRP). Recall that a permutation of a set X is a re-arranging of that set's elements. Or in other words, a permutation of a set X is a function $f : X \to X$ that is a bijection (both injective and surjective). So what does it mean to say that f is pseudorandom? For concreteness, say X is the set of all bitstrings of length n—and thus, it has 2^n elements. How many permutations are there from X to X? Well, for the first element of X, there are 2^n elements to take its place, for the second element of X, there $2^n - 1$, and so on, until we reach the 2^n-th element of X, for which there is one remaining choice. Hence there are $2^n!$ ways to permute the set X. Denote this set of permutations Perm_n (and note that $|\mathsf{Perm}_n| = 2^n! \gg 2^n$). Now consider the following two probability distributions over Perm_n: on the one hand, we have the uniform distribution, that chooses at random one permutation from Perm_n; on the other, we consider a *subset* of Perm_n, indexed by a set K, such that $|\mathsf{Perm}_n| \gg |K|$,

[12] This corresponds to Definition 3.33 given in ([9], p. 96).

and an algorithm that chooses a permutation from that subset, according to some probability distribution. Note that this latter case is also a distribution on Perm_n, albeit one where the permutations not belonging to the subset are assigned probability zero. We say that the second distribution is a **pseudorandom permutation** if no efficient distinguisher can, well, *distinguish* one distribution from the other. We shall be making these notions both more precise and more general in a moment, but for now note that the second case corresponds to the intuition of a block cipher. To see why, consider a *keyed pseudorandom permutation*: $F : K \times X \to X$, where if we fix a key, say k, then we obtain one of the permutations in the subset above: $F_k : X \to X$. Which is precisely what we want from a block cipher: that once a key is chosen, it behaves *as if* it was a permutation chosen uniformly at random from Perm_n—although in practice it cannot be.[13]

Remark 2.10 (Pseudorandomness of distributions). As the above discussion makes clear, the notion of pseudorandomness applies to *distributions*—not concrete objects, such as permutations or functions (see below). However, here we will follow a longstanding convention that allows us to abuse the language and speak of pseudorandom permutations or functions. △

Towards a more formal definition, first note that we do not need to confine our discussion to *permutations*, rather we can consider the more general case of (pseudorandom) *functions* (PRF)—the difference being that unlike a permutation, a function need not be a correspondence between the same set, and need not be injective and surjective, or equivalently, a function does not have to be invertible. Now the reader might wonder what is the point of a non-invertible block cipher—and while such a doubt is certainly legitimate, the notion of a PRF turns out to be quite useful. And even non-invertible blockciphers can be put to good use—cf. the *countermode* mode of operation (§2.5.3).

So let X and Y be (finite) sets, and consider a function $f : X \to Y$; how many such functions are there? Well, for each $x \in X$, there are $|Y|$ choices, and there are $|X|$ elements in X, which yields a grand total of $|Y|^{|X|}$. If X is the set of n-length bitstrings, and Y is the same but for length m, then the total number of functions is $(2^m)^{2^n} = 2^{m2^n}$. Denote this set by $\mathsf{Func}_{n,m}$. The intuition is similar to the above case: we consider keyed functions, $F : K \times X \to Y$, obtaining a subset of the functions in $\mathsf{Func}_{n,m}$, namely $F_k : X \to Y$. And we say that F_k is a pseudorandom function if an efficient adversary cannot distinguish the "behaviour" of F_k from that of a function f chosen at random from $\mathsf{Func}_{n,m}$.

Let D be an algorithm that purports to distinguish F_k from f. He is given access to an oracle, of either F_k or f—denoted respectively as $D^{F_k(\cdot)}$ and $D^{f(\cdot)}$. Suppose, without loss of generality, that D outputs 1 if he "thinks" that he is interacting with $D^{F_k(\cdot)}$, and 0 otherwise (i.e., $D^{f(\cdot)}$).[14] Then we have the following definition:

Definition 2.11. *Let $F : K \times X \to Y$ be an efficient keyed function. F is a* pseudorandom function *if for all PPT distinguishers D, there exists a negligible function* negl *such that:*[15]

$$\left|\mathrm{Prob}\left[D^{F_k(\cdot)}(1^n) = 1\right] - \mathrm{Prob}\left[D^{f(\cdot)}(1^n) = 1\right]\right| \leq \mathsf{negl}(n) \tag{2.9}$$

Note that D cannot be given the key k—for the process of how k indexes the subset of $\mathsf{Func}_{n,m}$ from where F_k is chosen is assumed to be known to D (cf. Kerckhoff's principle). And thus, if he knew k, he could choose some random x and compute himself $y = F_k(x)$. He then queries the oracle on x, obtaining answer y'. If he is interacting with $D^{F_k(\cdot)}$, then $y = y'$ with probability 1, which would only happen with probability 2^{-m} otherwise.

[13] Can you see why?

[14] Why is there no loss of generality here? I.e. why could we assign 1 and 0 in reverse?

[15] This corresponds to Definition 3.25 in ([9], p. 79).

2.4 PRNGs

I will give a simplified—but still formally correct—proof of Theorem 3.18 (Katz and Lindell [9], p. 68)—which in the book is more complicated than what it needs to be. Recall the definition of the security game $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$(previous section). We want to construct a weaker version of the one-time pad, but one which is still secure in practice—which in our context, means secure according to definition 2.4. The way we do this, is by replacing the ("true") random string (of length at least equal to that of the plaintext message) used in the OTP, by a *pseudorandom* string, which must also be of length at least that of the plaintext, but which can be generated from a much smaller amount of "true" randomness (the seed).

Formally, we use a *pseudorandom number generator* (PRNG) $G\colon \{0,1\}^n \to \{0,1\}^{l(n)}$, which expands a "true" random bit string of length n into a pseudorandom string of length $l(n)$.[16] Note that for an adversary with unbounded computational power, it is trivial to distinguish a uniform random string, from one that could have been output by G: just iterate over all the 2^n values of $\{0,1\}^n$, and check whether the given string is or not in the range of G. However, such a strategy is out of reach for efficient adversaries. We want G to be constructed in such a manner that, *for all strategies that **any** efficient adversary can execute*, the output of G "looks the same" as the output of choosing a string uniformly from $\{0,1\}^{l(n)}$. We capture this intuition in the following definition:

Definition 2.12. *Let l be a polynomial and let G be a* **deterministic** *polynomial-time algorithm such that, for any n and for any $s \in \{0,1\}^n$, $G(s) \in \{0,1\}^{l(n)}$. We say that G is a* **pseudorandom generator** *if:*[17]

1. ***Expansion:*** $\forall n, l(n) > n$.
2. ***Pseudorandomness:*** *for any PPT algorithm D, there exists a negligible function* negl *such that:*

$$\left|\mathrm{Prob}_{s\leftarrow\{0,1\}^n}[D(G(s))=1] - \mathrm{Prob}_{r\leftarrow\{0,1\}^{l(n)}}[D(r)=1]\right| \leq \mathsf{negl}(n) \tag{2.10}$$

The first probability is taken over the randomness of D, and the uniform choice of $s \in \{0,1\}^n$; the second over the randomness of D, and the uniform choice of $r \in \{0,1\}^{l(n)}$.

Note: for the sake of clarity, I may at times indicate in a subscript from which set a given value is sampled, as in a definition above ($r \leftarrow \{0,1\}^{l(n)}$ and $s \leftarrow \{0,1\}^n$). But usually I will omit it.

We now arrive at the desired modification of the OTP, which aims to make it practical: given plaintext message m, encipher it has $\mathsf{Enc}_k(m) = m \oplus G(k)$.[18] It is not known how to to prove that this construction—or any construction, for that matter—is unconditionally secure. What we can do, is argue in favour of the plausibility of the construction, assuming its underlying building blocks are secure. In the particular case of the construction above, we argue that it is secure, assuming the underlying pseudorandom generator, G, is secure. Katz and Lindell indeed do this (cf. [9], Theorem 3.18, p. 68), but in this case, there is a simpler—and perhaps more insightful—approach. We want to show that if G is secure, then so is Enc. We do this showing that if Enc is *insecure*, then G is also insecure.

[16] We shall always assume that the polynomial l *expands* the string, because from a practical point of view, one is usually not interested in reducing the amount of randomness one has (i.e. reducing the length of a "true" random string).

[17] This corresponds to Definition 3.14 given in ([9], p. 62).

[18] Note that the shared key of this symmetric encryption scheme is used as the seed of the PRNG. This corresponds to Construction 3.17 given in ([9], p. 67).

More concretely, assume there exists an adversary $\mathcal{A}$ that breaks Enc with probability $1/2+p(n)$, with $p(n)$ being **non-negligible**. That is, given two messages m_0, m_1, $\mathcal{A}$ can, just from the cryptogram, tell which message was enciphered (with probability $1/2+p(n)$). And if it guessed correctly, it outputs 1. We use this adversary to construct a distinguisher D that breaks G:

1. D receives a string $w \in \{0,1\}^{l(n)}$.
2. D executes $\mathcal{A}$, and obtains $m_0, m_1 \in \{0,1\}^{l(n)}$.
3. D chooses a bit $b \leftarrow \{0,1\}$, and constructs ciphertext $c = m_b \oplus w$.
4. D gives c to $\mathcal{A}$, obtains b from $\mathcal{A}$, and outputs 1 if and only if $b' = b$. Otherwise it outputs 0.

Recall we say that D *succeeds* when it outputs 1. It should be clear that if w is a truly random string, D succeeds with probability $1/2$ (because $m_b \oplus w$ is exactly the OTP), whereas if w is a pseudorandom string (output by G), D succeeds with probability $1/2+p(n)$:

$$\text{Prob}_{w\leftarrow\{0,1\}^{l(n)}}[D(w)=1] = \frac{1}{2} \tag{2.11}$$

$$\text{Prob}_{k\leftarrow\{0,1\}^{n}}[D(G(k))=1] = \frac{1}{2}+p(n) \tag{2.12}$$

We hence obtain:

$$\left|\text{Prob}_{w\leftarrow\{0,1\}^{l(n)}}[D(w)=1] - \text{Prob}_{k\leftarrow\{0,1\}^{n}}[D(G(k))=1]\right| \tag{2.13}$$

$$= \left|\frac{1}{2} - \left(\frac{1}{2}+p(n)\right)\right| = p(n) \tag{2.14}$$

As $p(n)$ is non-negligible, we just broke the pseudorandom generator G (cf. definition 2.12). Taking the contrapositive, we conclude that *under the assumption that G is a secure PRNG*, the encryption scheme outlined above is EAV-secure.[19]

An alternative proof. Instead of doing the contrapositive as above, we can consider an idealised version of the encryption scheme Π, which I will denote as $\widetilde{\Pi}$, which uses a message-length true random pad (instead of the pad generated by pseudorandom generator G).[20] Given a random string, any distinguisher (efficient or not) is correct with probability exactly $1/2$. This leads to the EAV-security condition for the OTP, namely that for any adversary (efficient or not), it succeeds in the $\text{PrivK}^{\text{eav}}_{\mathcal{A},\,\Pi}$experiment with probability exactly $1/2$. Thus, we can fix a distinguisher D constructed like above (and hence efficient), and a PPT (and hence efficient) adversary $\mathcal{A}$, for both of which it holds that:

$$\text{Prob}_{w\leftarrow\{0,1\}^{l(n)}}[D(w)=1] = \text{Prob}\left[\text{PrivK}^{\text{eav}}_{\mathcal{A},\,\widetilde{\Pi}}=1\right] = \frac{1}{2} \tag{2.15}$$

But due to the way D is constructed, we also have:

$$\text{Prob}_{s\leftarrow\{0,1\}^{n}}[D(G(s))=1] = \text{Prob}\left[\text{PrivK}^{\text{eav}}_{\mathcal{A},\,\Pi}=1\right] \tag{2.16}$$

[19] For those less familiar with formal logic, the strategy we followed here amounts show that $a \rightarrow b$, by actually proving the equivalent condition (called the *contrapositive*) $\neg b \rightarrow \neg a$.

[20] This means that in this case, our idealised construction $\widetilde{\Pi}$ coincides with the OTP.

And if G is a secure pseudorandom generator, then there must exist a negligible function negl such that:

$$\left|\text{Prob}\left[D(G(s))=1\right]-\text{Prob}\left[D(w)=1\right]\right| \leq \mathsf{negl}(n) \tag{2.17}$$

Substituting (2.15) and (2.16) into (2.17), we obtain:

$$\begin{aligned} &\left|\text{Prob}\left[D(G(s))=1\right]-\text{Prob}\left[D(w)=1\right]\right| && (2.18)\\ =&\left|\text{Prob}\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi}=1\right]-\text{Prob}\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\widetilde{\Pi}}=1\right]\right| && (2.19)\\ =&\left|\text{Prob}\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi}=1\right]-\frac{1}{2}\right| \leq \mathsf{negl}(n) && (2.20)\end{aligned}$$

The last inequality implies that:

$$\text{Prob}\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\,\Pi}=1\right] \leq \frac{1}{2}+\mathsf{negl}(n) \tag{2.21}$$

As the adversary was arbitrary, this shows that the scheme Π is EAV-secure.

Both proofs are formally correct, and hence equivalent. But I hope that comparing one to the other helps convince you that reasoning about reductions in terms of the contrapositive can yield a more enlightening argument.

2.5 Modes of Operation

For now we will only cover modes of operation of *block ciphers*. Hence, for the remainder of this section, assume we have a PRP F_k, that maps n bits to n bits. The constructions we will see below show how to apply it to input of arbitrary length.

2.5.1 Electronic Code Book (ECB)

This mode is presented merely for historical reasons, and because the practitioner of cryptography should be aware of it—**so it can be avoided like the plague**. Why? Because this mode of usage consists of simply chopping up the input into n-bit sized pieces, and applying F_k to each of them (of course this requires padding, but the mode is so insecure that discussing this becomes irrelevant). The reason for its insecurity should be easy to see: it is a *deterministic* scheme. I.e., ciphering the same message twice yields the same cryptogram, which means that at best, the scheme applied with a secure PRP can hope to be EAV-secure (and this is only if the size of the messages is restricted to the size of the block). But no matter how good the PRP, even MULT-security is not achievable.

2.5.2 Cipher Block Chaining (CBC)

This mode aims to correct ECB's flaw of determinism. To this end, it makes use of a *random Initialisation Vector* (IV). It is XOR'd with the first block of the message, prior to the call to F_k. This means that ciphering two messages with the same initial block, will *not* cause the first block of the cryptogram to be equal. And this latter first block will then be XOR'd with the next block of the plaintext prior to F_k, and so on. See figure 2.1. One difficulty of this mode, is that it is not easy to parallelise. I.e. to be able to compute ciphertext block n, one has to compute all previous ciphertext blocks. Another problem, is that as the IV is random, it has to be sent along with the

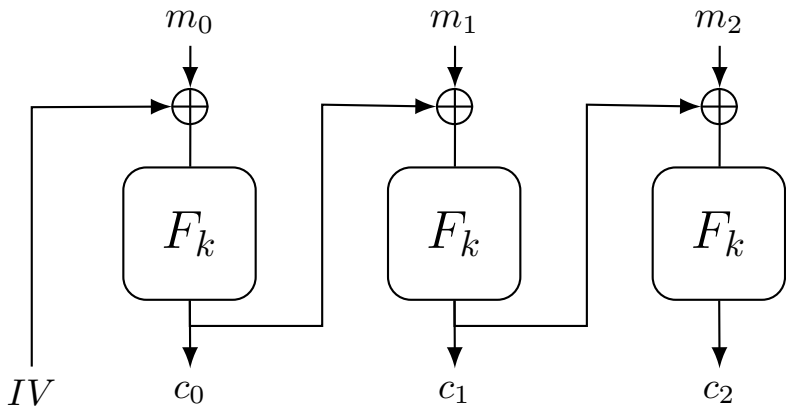


Figure 2.1: CBC encryption.

message. If the message size is small, this can be a significant increase (e.g. if the message size is the size of the block, then the ciphertext will be twice the length of the plaintext). Another limitation is that the IV *cannot be re-used*—otherwise CBC will leak information about the first block of the plaintext message. Lastly, note that with CBC, F_k has to be a (pseudorandom) *permutation*—because decryption requires that it be invertible. As we shall shortly see, this is not so with Countermode (CTR).

2.5.3 Countermode (CTR)

Here, the idea is to use F_k to produce a pseudorandom pad, to which the plaintext can then be simply XOR'd. For this reason, there is no need to pad the input. Furthermore, this method is highly efficient, not just due to its simplicity, but also because unlike CBC, it is very easy to parallelise. This is because to encrypt a given part of a message, one does not need to know the encryption of previous parts. See figure 2.2.

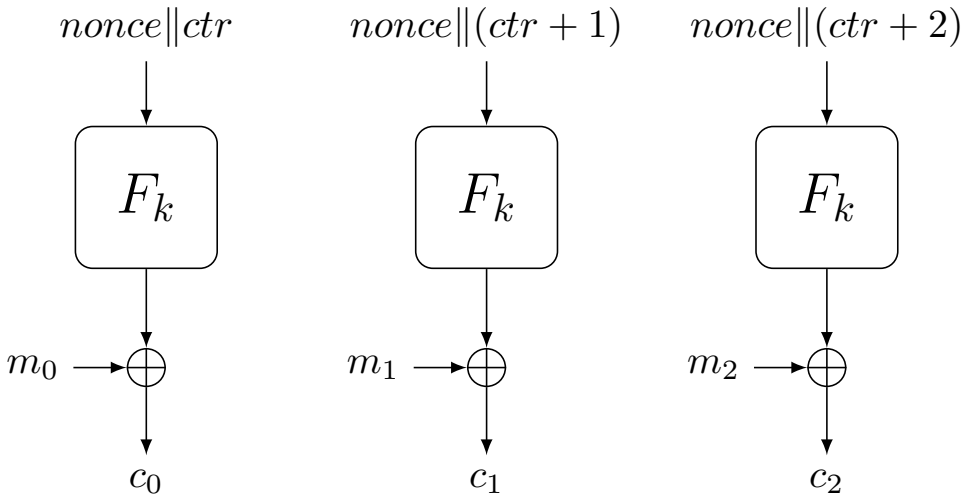


Figure 2.2: CTR encryption (based on [10]).

Also unlike CBC, here F_k does not need to be a *permutation*, for we do not need to invert it for decryption (why?). So it suffices for F_k to be a (possibly non-invertible) function—but obviously, **it still has to be pseudorandom!!**

In more detail, the way we generate a pseudorandom pad from F_k, is to take an unique nonce, and concatenate it with a counter. This guarantees that the input to F_k never repeats itself—and hence, from the pseudorandomness of F_k follows the pseudorandomness of the pad.

An important caveat. In CBC, if by mistake the IV is re-used, that may leak some information, but only regarding the first block of the plaintext (m_0). In CTR, however, **reusing the nonce re-generates the same pad**—which leads to a much greater leakage than what would happen in CBC. So it is of paramount importance never to re-use the nonce. This begin said, in practice this is relatively easy to ensure, so today the preferred usage mode for block ciphers is CTR.[21]

[21] Actually, as mentioned in several other places in the document, the preferred way to use block ciphers is to choose an AEAD mode (§3.5)—but if a block cipher mode must be used, CTR is the recommended choice.

3 Message Authentication Codes

3.1 Isn't secrecy enough?

Alas, no. Firstly, because depending on the situation, the integrity of the message may be more important than its confidentiality. For example, if you are doing a bank transfer, you might not care if the entire world knows about it—but you want to be darn sure that neither the amount of the transfer, nor its intended recipient, are modified! However, in scenarios where confidentiality is in fact required, It is intuitive to think that if an adversary cannot extract any information from a ciphertext, then he is also unable to modify said ciphertext in a meaningful way. The road to cryptographic wisdom, however, is filled with failed intuitions. To take an extreme example, consider that even the perfectly secret OTP does *not* provide integrity:[1]

Plain	HEILHITLER
Key	WCLNBTDEFJ
Cipher	DGTYIBWPJA

(a) Normal one-time pad usage.

Plain	HANGHITLER
Key	WCLNBTDEFJ
Cipher	DCYTIBWPJA

(b) But ciphertext modifications are impossible to detect.

Figure 3.1: Why the one-time pad offers no integrity guarantees.

But it gets worse. In fact, even if, in addition to a cipher with an appropriate usage mode (which gives you confidentiality), you use a primitive for integrity—i.e. a *message authentication code*, this chapter's topic—you can still end up with a construction that gets you neither of those goals. In fact, later in the chapter I will describe one such attack—the *padding oracle* attack.

3.2 Definitions

Before the definition, the idea: Alice wants to send message m to Bob, and in order to protect the message's integrity, she will *append* a tag t, which depends on both m and the secret key k she shares with Bob, and send both to Bob. On reception, Bob *recomputes* the tag, and if the result is equal to the tag t he received, then he accepts the message. Otherwise, he rejects it, and perhaps asks Alice to re-send it. Note that we are not trying to prevent an adversary from modifying the message; we are trying to prevent him from doing so *undetected*. Let us formalise this intuition.

[1] The student will recall that in a earlier lecture, I showed that if the key is destroyed after begin used, one can make the claim that the key actually had such a value that makes the ciphertext decrypt to any message we choose. This is the dual of that situation: instead of modifying the key, we modify the ciphertext.

Definition 3.1. *A* **message authentication code***, (or* MAC *for short), consists of three PPT algorithms,* $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$*, such that:*

1. *The key generation algorithm* Gen *takes the security parameter* 1^n*, and outputs a key* k *with* $|k| \geq n$*.*
2. *The tag generation algorithm* Mac *takes as input the key* k *and a message* m *of arbitrary length (i.e.* $m \in \{0,1\}^*$*) and outputs a tag* t*. This algorithm can be randomised, and hence it is denoted as* $t \leftarrow \mathsf{Mac}_k(m)$*.*
3. *The deterministic verification algorithm,* Vrfy*, takes as input a key* k*, message* m*, and tag* t*. The output is one bit:* 1 *if it considers the tag* valid*, and* 0 *if it considers the tag* invalid*.*

The **consistency condition** *requires that for every key output by* Gen*, and every message in* $\in \{0,1\}^*$*, it holds that* $\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1$*.*[2]

Canonical verification. In most practical algorithms, the verification step basically consists in recomputing the tag, and comparing it with what was received. This is called *canonical verification*. (Note, in passing, that canonical verification implies that the Mac algorithm is deterministic.) Conceptually, however, it is important to separate the semantics of authenticating a message from that of verifying that a message is authentic. Hence we consider distinct algorithms for each task.

Security of a MAC scheme. The intuitive idea is that the adversary must not be able to produce a valid tag on any "new" message—meaning a message that was not previously authenticated by an honest party. As usual we shall consider only computationally bounded adversaries, however we allow him access to a *MAC oracle*, $\mathsf{Mac}_k(\cdot)$, that produces tags for messages of his choosing. This is done to model the fact that the adversary may be able to influence what messages get Mac'd—just as in some private encryption settings (CPA and CCA security), he was assumed to be able to influence what messages get encrypted.

To lay these ideas down formally, consider the scheme $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$. We have the following experiment:

Message Authentication Experiment $\mathsf{Mac\text{-}Forge}_{\mathcal{A},\,\Pi}(n)$:

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Adversary $\mathcal{A}$ is given 1^n and oracle access to $\mathsf{Mac}_k(\cdot)$. Eventually $\mathcal{A}$ outputs (m, t).
3. $\mathcal{A}$ *succeeds* if and only if $\mathsf{Vrfy}_k(m, t) = 1$, and $m \notin \mathcal{Q}$, where $\mathcal{Q}$ is the set of messages on which the adversary queried the oracle. When this happens the output of the experiment is 1; otherwise it is 0.

We consider a MAC secure if no efficient adversary can win the above game, except with negligible probability:

Definition 3.2. *A message authentication code* $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *is* **existentially unforgeable under an adaptative chosen-message attack***, if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

$$\mathrm{Prob}\left[\mathsf{Mac\text{-}Forge}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \mathsf{negl}(n) \tag{3.1}$$

[2] This corresponds to Definition 4.1 given in ([9], p. 111).

When things are clear from context, we just say the MAC is *secure*.[3]

Remark 3.3. Note that this definition, albeit considerably strong, does not rule out things like *replay attacks*, where an adversary simply resends an already authenticated pair (m, t), over and again. Nor does it rule out the opposite, i.e. a secure MAC does not prevent an active adversary of causing messages to "disappear". Or the possibility that messages get reordered. The reason is that when we design cryptographic primitives, we do so *without making any assumption about the applications that will use them*. And whether replay attacks or disappearing or reordered messages are a problem or not, is a question that fits squarely within the semantics of the application where a MAC is being used. △

Strongly secure MAC. Another thing that the adversary, in the above definition, is not precluded from doing, is taking an already authenticated message m, and compute *another* valid tag for that message. That is, given (m, t), compute t' such that $\mathsf{Vrfy}_k(m, t') = 1$. If adversary $\mathcal{A}$ is able to do this with MAC scheme Π, that still does **not** allow him to win the $\mathsf{Mac\text{-}Forge}_{\mathcal{A},\,\Pi}$ security game, with probability better than negligible—and thus Π is still considered secure, under that definition. In practice this not usually a concern—after all, if a message is already authenticated, computing a new valid tag just re-states that fact. There are, however exceptions—we will see an important one below, when discussing the *encrypt-then-mac construction*—and to rule out this behaviour, we modify experiment Mac-Forge by changing $\mathcal{Q}$: instead of containing just a list of messages for which the adversary queried the oracle, it will also contain the *tags* the oracle returned. That is, $(m, t) \in \mathcal{Q}$ if $\mathcal{A}$ queried the oracle on message m, and got tag t as response. We denote this new experiment as Mac-SForge, and say that a MAC scheme as defined above is **strongly secure** if no efficient adversary wins this new experiment with more than negligible probability (this is nothing more than replacing experiment Mac-Forge with Mac-SForge in definition 3.2). In this new security game, if the adversary can request a tag for message m, and from the pair (m, t) (where t is the tag he got from the oracle), he can concoct another tag t' for the same message m, then here he **is** allowed to submit (m, t') as his forgery—as $(m, t) \in \mathcal{Q}$, but $(m, t') \notin \mathcal{Q}$. If the verifier accepts this new tag, then he wins the game. So to say that no efficient adversary can win the Mac-SForge security game, with probability better than negligible, is effectively to rule out the possibility the adversary being able to produce new tags on previously authenticated messages.

Obviously, this new game Mac-SForge also rules out the possibility of the adversary discovering a tag a for a new message—that is, strong security implies ("regular") security. The next theorem establishes a condition for the reverse to also hold.

Theorem 3.4. *Every secure MAC that uses canonical verification is strongly secure.*

Proof. As stated above, canonical verification implies a deterministic Mac algorithm, which means that if (m, t) is a valid pair, then for any $t' \neq t$, (m, t') is an invalid pair (i.e. rejected by the verifier). Hence, to win the Mac-SForge game, the adversary must forge a tag on a new message—which is what he had to do to win the Mac-Forge game. The result now follows. ■

What about a verification oracle? One can argue that a more "natural" way to model a MAC scheme is by also allowing the adversary, in addition to obtaining tags for messages of his choice, is also to *verify* pairs (m, t) of his choice. This can be easily done by adding a verification oracle $\mathsf{Vrfy}_k(\cdot, \cdot)$ to experiments Mac-Forge and Mac-Forge. However, for canonical verification MACs (which includes all those used in practice), it turns out that $\mathsf{Vrfy}_k(\cdot, \cdot)$ is basically useless. This

[3] This corresponds to Definition 4.2 given in ([9], p. 113).

is because given a pair (m, t), the adversary either previously queried m to $\mathsf{Mac}_k(\cdot)$ (and thus knows the answer that $\mathsf{Vrfy}_k(m, t)$ will give him), or didn't previously queried m to $\mathsf{Mac}_k(\cdot)$—and in this case he also knows, from the security of the MAC (strong or otherwise, no matter[4]), that $\mathsf{Vrfy}_k(m, t) = 0$, with overwhelming probability (only negligibly smaller than 1).

Timing attacks.[5] All the theory in the world will not help if when checking a MAC tag, you do it byte by byte, and—if the tag you compute is different from the one you got—**you output invalid as soon as you find that difference**. The reason is that an attacker can try all the 256 possibilities for the first byte of a random tag, and guess that the correct value is the one where it took longer for invalid to be output—because the next byte would then have had to be checked. And then do the same for that next byte, and so on. Thus if you have, say, a 16-byte tag, you only need $256 \times 16 = 4096$ attempts, instead of 256^{16} attempts that an exhaustive search would require. For reference:

$$256^{16} = 340282366920938463463374607431768211456 \approx 3.4 \times 10^{38} \tag{3.2}$$

In practice, even time differences of the order of milliseconds was enough to break security. So bottom line: **always check all the bytes of the tag, so it always takes the same time!**

3.3 Constructing MAC schemes

If we restrict its input to messages of a fixed length, say n, then any pseudorandom function with input of size n can be used as a MAC. The security of the MAC follows from the pseudorandomness of the function: it is impossible to guess the tag for a given (new) message with probability better than $2^{-n} \pm \mathsf{negl}$. Note that in this case, the MAC is also strongly secure, for there is only one tag for any given message.

In this approach, however, the messages besides being limited to a fixed length, will also be very small, as the input length of practical pseudorandom functions is usually around a few hundred bits. Katz and Lindell [9] give one elegant approach to overcome this length limitation—i.e. to perform *domain extension* for (fixed length) MACs (§4.3.2). But while elegant, that construction is also not very efficient.

A more promising approach might be to take the efficient constructions we do have for pseudorandom functions (in particular, block ciphers, but here I consider a generic pseudorandom function F), and try to turn them into an efficient MAC schemes. For example, we can tentatively define a CBCMAC scheme, based on the CBC mode of operation for block ciphers, as shown in figure 3.2. So we ignore the IV (or equivalently, set it to a block of zeroes), and do not output any intermediate "ciphertext" blocks, only the last one—and that will be our tag for message $m = m_0 \| m_1 \| m_2$. This scheme is secure as long as **all the messages are of fixed length**. The length itself is arbitrary, but it must be agreed beforehand by both sender and receiver. To get the intuition of what goes wrong if messages of different length are allowed, consider a message of length equal to the input size of F_k, call it m_0. The we obtain from the CBCMAC version depicted in figure 3.2 is simply $t = F_k(m_0)$—in fact, this reduces to using the PRF F as a MAC. But if we add another block $m_1 = m_0 \oplus t$, then we can compute, without needing the secret key k, the tag for the new message $m_0 \| m_1$: in fact, it will be t again! (Exercise: check this.[6])

[4] Remember both security notions are equivalent under canonical verification.

[5] Timing attacks are one instance of what is generally known as *side-channel attacks*. These work by trying to extract (secret) information from "side-channels": time measurements, power measurements, electromagnetic measurements, …

[6] Another exercise: produce a forgery for the example depicted in figure 3.2. Hint: you need another three blocks.

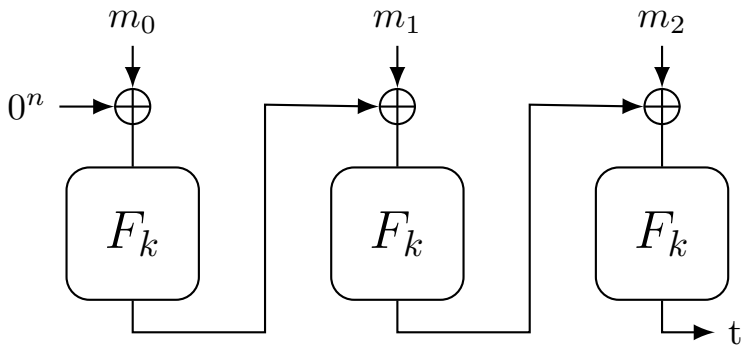


Figure 3.2: CBCMAC for fixed length inputs. Modified from [4].

It turns out that the problem that breaks security of this scheme with messages of different lengths, is that they can share a *common prefix*. There are two ways of solving this. One is instead of returning tag t in figure 3.2, use another key k', **chosen independently of** k, and return tag $\hat{t} = F_{k'}(t)$. But this requires generating another key, and as key generation is usually considered the "slower part", broadly speaking, of symmetric constructions, one usually prefers avoiding extra keys. So another approach is simply to modify the message to ensure that prefixes are impossible: this is achieved simply by prepending to message m, its length $|m|$. That is, to calculate the tag for m, we run CBCMAC (now allowing for length variability) on message $|m| \,||\, m$. I stress that the length has to be **prepended**, that is, inserted at the beginning, otherwise prefixes could still occur. This latter strategy has the disadvantage of needing to know the length of the message when beginning to compute the MAC, but this is usually preferable to having to use a second key (but again, this depends on the concrete usage scenario.)

3.4 Authenticated Encryption

As the padding oracle attack described in the next section will show, it is a huge mistake to assume that just confidentiality is enough to prevent an adversary from getting to mischief. In fact, it is even worse than that: that attack showed that even using constructions for **both** confidentiality and integrity, if they are combined without due care, disaster can ensue.

In more detail, let us try to define what we are trying to achieve. What are we trying to achieve? Well, we want a construction that gives us both confidentiality and integrity. Now, worrying about integrity makes no sense unless we consider an adversary that is capable of actually *modifying* the data sent from one honest party to another. So, in what confidentiality is concerned, we want CCA-security (§2.2). How should integrity fit into the mix? Well, we just replicate (actually, adapt) the existentially unforgeability experiment.

Unforgeable Encryption Experiment $\mathsf{Enc\text{-}Forge}_{\mathcal{A},\,\Pi}(n)$:

1. Run $\mathsf{Gen}(1^n)$ to generate a key k.
2. Adversary $\mathcal{A}$ is given 1^n and oracle access to $\mathsf{Enc}_k(\cdot)$. Eventually $\mathcal{A}$ outputs a ciphertext c.
3. Let $m = \mathsf{Dec}_k(c)$. $\mathcal{A}$ *succeeds* if and only if $m \neq \perp$, and $m \notin \mathcal{Q}$, where $\mathcal{Q}$ is the set of messages on which the adversary queried the oracle. When this happens the output of the experiment is 1; otherwise it is 0.

Here we introduce a new behaviour for the decryption algorithm: it can *refuse* to decrypt a ciphertext, instead outputting what can, for the purposes of the present text, be thought of as a "rejection sign." It does not belong to the set of plaintext messages.

Definition 3.5. *An encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **unforgeable***, if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

$$\mathrm{Prob}\left[\mathsf{Enc\text{-}Forge}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \mathsf{negl}(n) \tag{3.3}$$

Remark 3.6 (No decryption oracle?). The attentive reader will have noticed that, even though we are trying to give definitions for CCA-resistant constructions, in the above experiment the adversary is not provided with a decryption oracle. It is possible to strengthen this model by adding one such oracle, but the given experiment is the classical way to define unforgeability (Katz and Yung [8]). The intuition here is that if *«any (new) ciphertexts generated by an adversary are likely to be invalid, the adversary cannot gain much by submitting them to the decryption oracle»* (ibid., §3.2). If the decryption oracle gave some hint of a *reason* for why the decryption failed, it could be argued that that information could help the adversary in producing a ciphertext that would decrypt correctly. But since the decryption oracle will almost always simply output $\perp$, it really does not help the adversary's task. △

Note that $\mathsf{Enc\text{-}Forge}$ does not imply CCA-security! In fact, considering the encrypt-then-mac construction (see below), with a MAC that is secure, but not strongly secure, is unforgeable. But as we will see, **it is not CCA-secure!**

Combining unforgeability and CCA security yields authenticated encryption:

Definition 3.7. *A private-key encryption scheme that is unforgeable and CCA-secure is an* **authenticated encryption scheme***.*[7]

This is the property that we want our cipher + mac constructions to have!!

We will see three approaches to combine a MAC and symmetric cipher, which I shall call encrypt-and-mac, mac-then-encrypt, and encrypt-then-mac. However, in practice, if you can, **you should not use any of those!** What you should do instead, is use modes like `AES-GCM` (AES in Galois Countermode), which provide you with both confidentiality and integrity, "for free", as it were. See §3.5. We will study those three approaches, however, because from a security perspective, it is often as important—if not *more* important—to know what you should *not* do, as to know what you *should* do.

Before proceeding, however, I again remark what was said in §1.3: in what follows, we always use **two independent keys**—I shall denote them k_E for the key used in the encryption scheme, and k_M for the key used in the MAC. To quote directly from Katz and Lindell, *«different instances of cryptographic primitives should always use independent keys»* ([9], p. 140).

Encrypt-and-mac. In this mode we encrypt message m, then compute the tag for message m, and send both: $\mathsf{Enc}_{k_E}(m) \| \mathsf{Mac}_{k_M}(m)$. **Avoid it at all costs.** The reason is that it is possible to have a secure MAC, that nonetheless produces a tag that is *correlated* with the message. Which is precisely what the encryption of m tries to avoid! So this defeats the purpose of encryption. But it gets worse. If the MAC used is deterministic, as are most of those used in practice, then seeing the same tag twice tells the adversary that the same message was sent twice (note that the tag is sent in the clear). So this mode may not achieve even CPA-security, and so is best avoided.

[7] This corresponds to Definition 4.17 in ([9], p. 132).

Mac-then-encrypt. In this mode, we first compute the MAC of the message, $t \leftarrow \mathsf{Mac}_{k_M}(m)$, concatenate the message with it, and encrypt the bundle: $\mathsf{Enc}_{k_E}(m\|t)$. This has intuitive appeal, as all the structure is hidden inside the ciphertext. However, this is only an illusion; in fact, the padding oracle applies to exactly this way of joining a block cipher with a MAC. Moreover, while this construction can be made secure, this is done imposing restrictions that go well beyond the cryptographic sphere, and into the design of the application: in particular, when decryption fails, it is required to hide the reason why it happened (invalid MAC or invalid pad; cf. §3.6). This may hinder the usability of the application, or make debugging harder, etc. In other words, it is not possible to analyse the security of this construction while abstracting away details of how it is (or will be) used. Hence, this approach is also best left unused.

Encrypt-and-mac. Here we first encrypt the plaintext, then compute the MAC of the resulting ciphertext. So we have the ciphertext, $c_m \leftarrow \mathsf{Enc}_{k_E}(m)$, and then we compute the tag over it, $t \leftarrow \mathsf{Mac}_{k_M}(c_m)$. And then we send both, i.e. the pair $c = (c_m, t)$.

Terminology: a word of warning. The reader may have noticed that above, that the ciphertext resulting from encryption was denoted c_m. This is because, as will happen below in the analysis of security, when interacting with an encryption or decryption oracle that uses this construction, the output or input of that oracle—the pair (c_m, t)—*will also be called ciphertext!* Usually this will not cause any trouble, as it will be easy to disambiguate from context—but the reader should be aware that this happens.

Going back to encrypt-then-mac, this mode is secure—meaning it provides authenticated encryption—*if the MAC used is strongly secure*, and the cipher is ***CPA**-secure*. To see why, let $c = (c_m, t)$ be a ciphertext that the adversary got from his encryption oracle. The "regular" security of the MAC implies that if he changes c_m, he cannot compute a valid tag for it—and so if he tries to decrypt that modified ciphertext the adversary will get $\perp$. This shows the scheme is unforgeable. To see that it is also CCA-secure, note that the strong security of the MAC makes decryption oracle in the CCA experiment useless: if the adversary makes a query with a ciphertext he got from the encryption oracle, he already knows the result; if he makes a query with any other ciphertext, the result will be an error ($\perp$). So CCA-security of the encrypt-then-mac scheme reduces to the CPA-security of the encryption scheme being used.

Observe that if the MAC scheme was secure, but not strongly secure, then:

- The scheme would still be unforgeable. For even if the adversary were able to produce a new tag for c_m, the decryption of c_m (i.e. m), is an already observed plaintext—it was the plaintext the adversary sent to the encryption oracle to obtain $c = (c_m, t)$.
- The scheme *would no longer have CCA-security* (and so it would no longer be a authenticated encryption scheme). To see why this, let $c = (c_m, t)$ be the challenge ciphertext (cf. §2.2). If the adversary can produce a new tag for c_m, then he obtains *a new ciphertext* $c' = (c_m, t')$, **which can be submitted to the decryption oracle!** Thus he will know, with probability 1, if c is the encryption of m_0 or m_1—and so this completely destroys CCA-security.

3.5 Authenticated Encryption with Associated Data

The previous section, combined with the padding oracle attack (next section), should suffice to convince you that whenever you need confidentiality, you should use a construction that also gives you integrity assurances.[8] And as mentioned in the previous section, the recommended

[8]The converse is of course false. If you just need integrity, there is no need to bother with confidentiality, just use a MAC.

way to do that, is not to use any of the constructions studied therein, but to use a cipher that provides you **authenticated encryption with associated data (AEAD)**. That is, it provides you with confidentiality and integrity, "for free"—meaning you do not have to do anything besides using said cipher. In fact, besides specifying the data to be encrypted, you can optionally specify additional data for which you just want integrity; i.e. additional data that gets sent in the clear, but for which the construction provides integrity guarantees.

Here I will briefly illustrate usage of the AESGCM construction (basic familiarity with the Python library `cryptography` is assumed).[9] Encryption works as follows:

```
# Generate a random 96-bit IV.
iv = os.urandom(12)
# And a 256-bit key.
key = os.urandom(32)

encryptor = Cipher(algorithms.AES(key), modes.GCM(iv)).encryptor()

# Associated_data, if any, will be authenticated but not encrypted.
# It must also be passed in on decryption.
if associated_data is not None and len(associated_data) > 0:
  encryptor.authenticate_additional_data(associated_data)

ciphertext = encryptor.update(plaintext) + encryptor.finalize()

return (iv, ciphertext, encryptor.tag)
```

As for decryption, we have:

```
decryptor = Cipher(algorithms.AES(key), modes.GCM(iv, tag)).decryptor()

# If it is not None, we put associated_data back in, or the
# tag will fail to verify when we finalize the decryptor.
if associated_data is not None and len(associated_data) > 0:
  decryptor.authenticate_additional_data(associated_data)

return decryptor.update(ciphertext) + decryptor.finalize()
```

See the documentation for other AEAD possibilities, such as `ChaCha20Poly1305`.

3.6 Padding oracle

Consider again CBC-encryption, using AES, for a three-block message $m = m_0 \| m_1 \| m_2$ (see figure 3.3), and using the **mac-then-encrypt** construction. That means that m consists of a plaintext message pt, for which a (MAC) tag t was computed, and the concatenation of both was padded to obtain a length multiple of the AES block size (16 bytes). That is, m can also be written as $m = pt \| t \| pad$. The pad scheme is simply: if p bytes are needed, then we use p bytes each containing the value p (written in binary). If the length of $pt \| t$ already is a multiple of the block size, then we use an extra block, containing the byte `0x16`, 16 times. Note that

[9] https://cryptography.io.

in figure 3.3, the pad will be in m_2—but we can ignore this and aim for recovering any other block; here we will target m_1.

To that end, consider the *leftmost* byte of both m_1 and c_0. We have (I am using Python-style array notation, where `[-1]` refers to the rightmost byte):

$$m_1[-1] = c_0[-1] \oplus \mathrm{AES}_k^{-1}(c_1)[-1] \tag{3.4}$$

To recover the last byte of m_1, we simply discard all subsequent blocks (in this case it would be just m_2 and c_2), so that m_1 now becomes the last block. And we rewrite (3.4) as:

$$m_1[-1] \oplus g \oplus \mathtt{0x01} = c_0[-1] \oplus g \oplus \mathtt{0x01} \oplus \mathrm{AES}_k^{-1}(c_1)[-1] \tag{3.5}$$

Mathematically, we just added $g \oplus \mathtt{0x01}$ to both sides; but it terms of figure 3.3, this means we replace the last (i.e. rightmost) byte of c_0, with that same value, XOR'd with $g \oplus \mathtt{0x01}$. That is, we replace $c_0[-1]$ with $c_0[-1] \oplus g \oplus \mathtt{0x01}$.

Now g is simply a guess for the value of byte $m_1[-1]$. To make the discussion more concrete, suppose that $m_1[-1] = \mathtt{0xAF}$. If g is different from this value, then the last byte of the decryption of our modified c_0 block will be:

$$\mathtt{0xAF} \oplus g \oplus \mathtt{0x01} \tag{3.6}$$

With overwhelming probability, this will not produce a valid pad (remember that m_1 is now the last block of the message). And as we are using mac-then-encrypt, in reverse, decryption happens first, and it is during decryption that the pad is checked—and if is invalid, it leads to a decryption error.[10] But consider what happens if $g = \mathtt{0xAF}$. If this is so, then (3.5) simply becomes:

$$\mathtt{0x01} = c_0[-1] \oplus \mathtt{0xAF} \oplus \mathtt{0x01} \oplus \mathrm{AES}_k^{-1}(c_1)[-1] \tag{3.7}$$

In other words, our modified c_0 decrypted to an m_1 block that ends with the byte `0x01`—**and this is a valid pad**. Which means that the receiver will move to the next step, verifying the MAC tag—which will fail, with overwhelming probability, as we just modified the plaintext. But it does not matter: the value of g for which we get a MAC error, will be be the last byte of the *original* m_1 plaintext block. Hence, we have recovered one byte of plaintext, at the cost of iterating g for only the 256 possible values for a byte.

[10] Depending on the implementation, the error returned may be a more specific *padding error*. But this is irrelevant, as long as it is different from a MAC error.

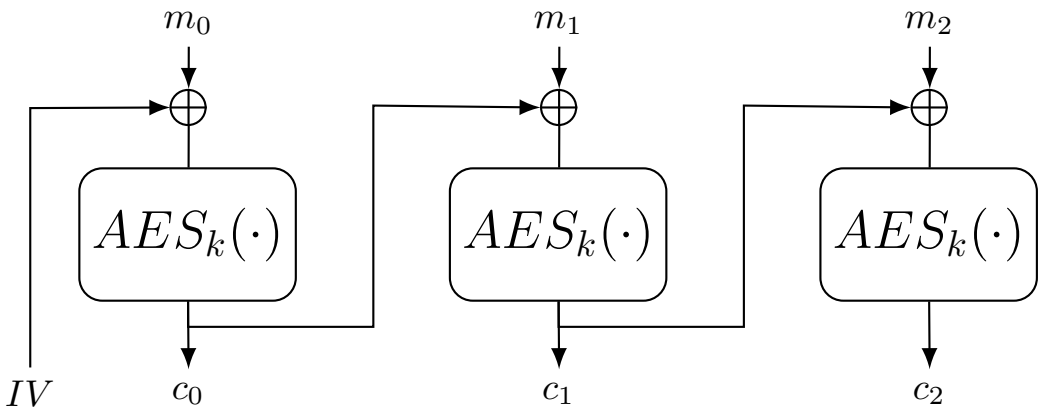


Figure 3.3: CBC encryption, revisited.

Now to recover the next byte of m_1, i.e. $m_1[-2]$, we do the same trick, but using the pad `0x02 0x02`. As `0xAF ⊕ 0xAD = 0x02`, we replace $c_0[-1]$ with $c_0[-1] \oplus \texttt{0xAD}$—this ensures that $m_1[-1]$ will be `0x02` (cf. (3.4)). And rewrite (3.5) for $c_0[-2]$ and $m_1[-2]$:

$$m_1[-2] \oplus g \oplus \texttt{0x02} = c_0[-2] \oplus g \oplus \texttt{0x02} \oplus \texttt{AES}_k^{-1}(c_1)[-2] \tag{3.8}$$

And again, iterate over the 256 possible values of g, until the decryption/padding error turns into a MAC error, at which point we obtained another byte of m_1, namely $m_1[-2]$. This strategy can be continued until all of m_1 has been recovered. And of course, this can be done for blocks other than m_1…

Mitigation. One obvious way of mitigating this problem, is for the receiver to always return the same error, regardless of what happens. However, even this has to be implemented carefully: if the verification of the pad plus MAC tag is not made in **constant time**, this still leaves a padding oracle. Why? Because an error due to bad padding will systematically take less time to be returned, than an error due to a bad MAC. Cf. the remarks concerning **timing attacks**, made at the end of §3.2, in particular the reduction from 256^{16} to merely 256×16 attempts. They apply here verbatim.

Of course, the ideal strategy, when one is, e.g., designing the protocol from scratch, is to use an AEAD mode (§3.5), such `AES-GCM` or `ChaCha20Poly1305`—but in practice, especially with systems that are already largely deployed, this might not be possible (usually in such scenarios, we have to keep changes to a minimum…).

4 Hash Functions

4.1 Intuition

The intuition of hash functions is that they map inputs of arbitrary length ($\{0,1\}^*$) to outputs of some fixed length n ($\{0,1\}^n$). This necessarily implies that collisions *must* exist; in other words, it is impossible for such a function to be injective. (Note that any injective function is trivially collision resistant.) For cryptography, however, we have the additional requirement that collisions must be very hard to find.

We will see below what this means more precisely, but it is important to keep in mind that the requirements for cryptography are more stringent that in other settings where hash functions may be used. For example, in data structures, it may suffice to have an hash function for which it is very unlikely to find collisions by chance. In cryptography by contrast, we need collisions to be hard to find, even when one is specifically looking for them. This makes cryptographically secure hash functions much harder to design.

4.2 Definitions

Although in practice cryptographically secure hash functions are unkeyed, for theoretical reasons they are modelled taking as input a key s, in addition to the message m over which the hash is supposed to be computed. This value (the output of the hash function) is also called the *digest* of message m. The reason for key s is that without it, H is just some fixed function, and there is always a constant time algorithm that outputs a collision—i.e. a pair (x, x') such that $H(x) = H(x')$—namely an algorithm that simply has one such pair hard-coded. If the reader finds this somewhat pedantic, this author sympathises, but that is how it is. However, this does mean that there are important differences between keys for hash functions, and the keys used in symmetric primitives studied so far. First and foremost, the latter have to be *secret*, while the former don't. For this reason, an hash function with key s is denoted H^s (instead of H_s, which was the usual notation for symmetric primitives). Secondly, for symmetric primitives any value of the key was valid; here however there may exist values of s for which H^s is not defined.

So even though we ignore the key in practice, we still define an hash function as a pair of algorithms:

Definition 4.1. *An* **hash function** *is a pair of PPT algorithms* (Gen, H)*, satisfying:*

1. Gen *is a probabilistic algorithm, which receives the security parameter* 1^n*, and outputs a key* s*.*
2. H *takes a string* $x \in \{0,1\}^*$ *and outputs a string* $H(x) \in \{0,1\}^{l(n)}$*, where* $l(n)$ *is some length value that depends on the security parameter* n*.*

It is possible to have an hash function that instead of receiving inputs of arbitrary length, receives only inputs of a given fixed length. In this case, we care only about hash functions for

which the input length is greater than the length of the digest $l(n)$—and when this is so, we say that the hash function is a **compression function**.[1]

We define security of the hash function $\Pi = (\mathsf{Gen}, H)$ by defining, given an adversary $\mathcal{A}$ and the security parameter n, the following collision finding experiment.

The collision-finding experiment $\mathsf{Hash\text{-}coll}_{\mathcal{A},\,\Pi}$:

1. Running $\mathsf{Gen}(1^n)$ yields a key s.
2. $\mathcal{A}$ is given s and outputs x, x'.
3. The output of the experiment is 1 if and only if $x \neq x'$ and $H^s(x) = H^s(x')$. If this is so, we say that $\mathcal{A}$ has found a collision.

The function H is said to be *collision resistant* if no efficient adversary has a meaningful probability of winning this game:

Definition 4.2. *An hash function* $\Pi = (\mathsf{Gen}, H)$ *is* **collision resistant** *if for all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that:*

$$\mathrm{Prob}\left[\mathsf{Hash\text{-}coll}_{\mathcal{A},\,\Pi}(n) = 1\right] \leq \mathsf{negl}(n) \tag{4.1}$$

This is the strongest security property that an hash function can have; however for certain applications, weaker notions may suffice. In particular, we have the following two:

Second preimage resistance: this means that given x, the adversary is not able to discover a different x' such that $H^s(x) = H^s(x')$.

(First) preimage resistance: given a digest y, the adversary is not able to compute x such that $H^s(x) = y$.[2]

Note that if an adversary can break preimage resistance, he can also break second preimage resistance. To see why, note that if he has an algorithm to, given y, compute an x for which $H^s(x) = y$, then he can break second preimage resistance as follows: take the given x, and compute $y = H^s(x)$. Now give y to the previous algorithm, which will output an x' such that $y = H^s(x')$. As the domains of hash functions, even if finite, are very large, chances are that $x \neq x'$—and so second preimage resistance has been broken.

Similarly, it is possible to show that an adversary breaking second preimage resistance can break collision resistance (exercise). So we have the following chain of implications (but note that the converses are false!):

Collision resistance → Second preimage resistance → Preimage resistance

4.3 Domain Extension: Merkle-Damgård

A construction that is extensively used in practice, is the *Merkle-Damgård* construction. It works on any fixed-length compression function, by turning it into a variable length compression function. Say h^s compresses its input from length $2n$ to length n. Then the transform yields a variable length hash function H^s as follows (note that the key s remains the same):

[1] If the number of possible inputs is smaller than the number of possible digests, then collision resistance is trivial to achieve (injectivity).

[2] Roughly, this means that H is a *one-way function*.

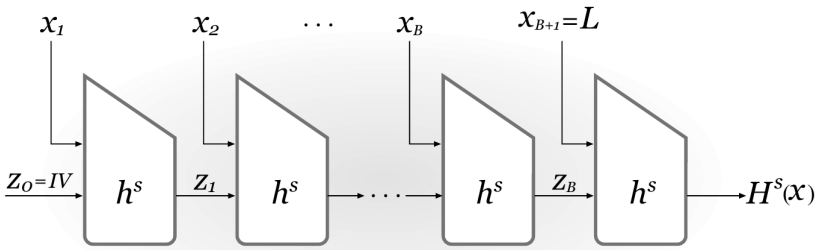


Figure 4.1: The Merkle-Damgård transform. Borrowed from Katz and Lindell [9], p. 158.

1. In input x, pad it with zeroes until you get a length that is multiple of n. Say this yields B blocks. Define block x_{B+1} to be equal to the bit representation of L, where L is the original (unpadded) length of input x.
2. Set $Z_0 = 0^n$, and $Z_i = h^s(Z_{i-1} \| x_i)$. See figure 4.1.[3]
3. Output $H^s(x) = Z_{B+1}$.

The security of the construction follows from the fact that if a collision is found for H^s, then that allows us to discover a collision for h^s. And so, if h^s is collision resistant, so will H^s. The intuition is easy to grasp.[4] Say you found a collision for H^s, i.e. x and x' such that $H^s(x) = H^s(x')$. Let L and L' be the lengths of x and x' respectively. Then we have two possibilities:

- $L \neq L'$. Then $Z_b \| L \neq Z_b \| L'$, and the last iteration of h^s shows a collision for it.
- $L = L'$. Then, given that $|x| = |x'|$ but $x \neq x'$, there must exist i such that $Z_i \| x_{i+1} \neq Z_i \| x'_{i+1}$. The inputs to h^s corresponding to the *largest* such i will yield a collision for h^s, as from that point onwards the values will be equal (otherwise we are not considering the *largest* such i).

Extension attacks. As the function h^s is publicly known, an adversary can take the output $H^s(x)$, and compute $h^s(H^s(x) \| y)$, where y is some value of the adversary's choosing. The next section shows a construction where this turned out to be a major problem.

4.4 HMAC

Before turning to HMAC proper, consider a MAC for variable length messages constructed in the following manner: given a collision-resistant hash function H^s, and a secure MAC Mac, to compute the tag of a message m, do $\mathsf{Mac}_k(H^s(m))$. Intuitively, this should be secure, because the collision resistance of H^s means that the digest $H^s(m)$ can, for practical purposes, be used as a replacement for m that is "as good as m". I.e., it is infeasible to find another message m' such that $H^s(m') = H^s(m)$. And indeed this construction, called *hash-and-MAC*, is secure—see ([9], §5.3.1) for a formal proof.

The approach described above is not (directly) used in practice. Another construction, standardised as **HMAC**, is used instead. It was first used with hash functions constructed via the Merkle-Damgård construction (e.g. MD5 or SHA256), and it can be seen as an "instantiation" of the more general hash-and-MAC approach—thus can be used with hash function H^s.

[3] Z_0 can be any fixed, public value, although it is customarily set to the zero vector.
[4] See Theorem 5.4 in ([9], §5.2) for a more detailed discussion.

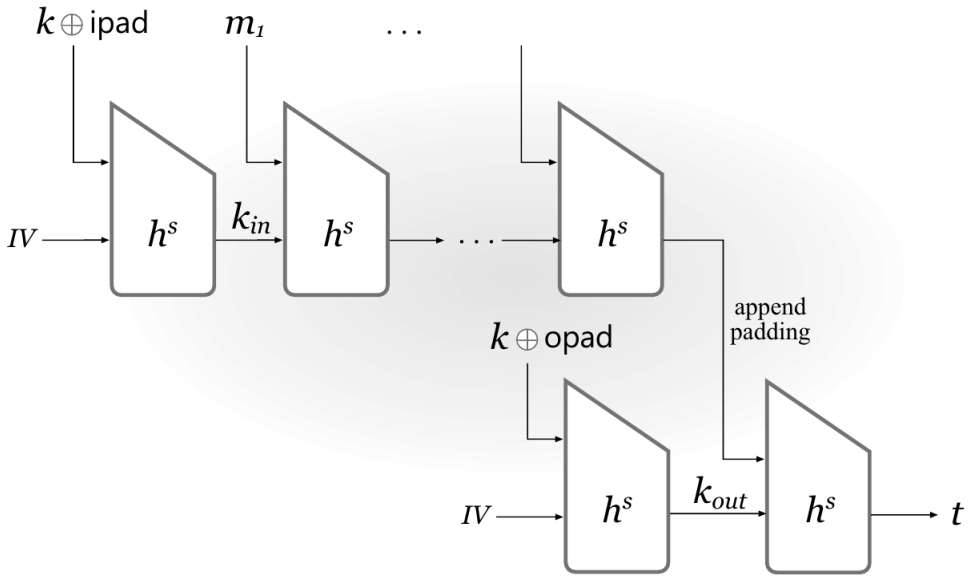


Figure 4.2: The HMAC construction. Borrowed from Katz and Lindell [9], p. 162.

Besides being standardised, and being *fast*—one of the reasons why practitioners avoided CBCMAC was because it was "slow"—it provides another benefit: it is secure even if the hash function used satisfies only a weaker form of collision resistance—creatively named *weakly collision resistance*—where the adversary is still trying to find a collision, but now with the aid of an "hash oracle" that allows him to obtain hashes for values of his choice, but with the function **secretly keyed** with k_{in}.[5] This turned out to be of major importance in practice, because one of the first functions to be used with this construction was MD5 (denoted HMAC-MD5). When the first collision attacks were discovered for MD5, it could no longer be considered collision resistant. However, it still verified the condition of weakly collision resistance, which meant HMAC's security proof was still valid. This still meant existing systems should, in the interest of caution, start using a better construction than HMAC-MD5, but the change could be properly planned, as there was no need to rush.

HMAC. The construction is depicted in figure 4.1. For now, think of $k \oplus \mathsf{ipad}$ and $k \oplus \mathsf{opad}$ as independent keys k_1 and k_2—we will get back to them momentarily.[6] To make sense of the diagram, ignore k_{in} and k_{out}, and think of the top "row" as an application of the Merkle-Damgård construction to $k_1 \| m$. The reason we cannot just use the top row as a MAC, i.e. given the secret key k_1, just compute Merkle-Damgård of $k_1 \| m$, is because this is vulnerable to *extension attacks*.[7]. Indeed, consider a message m, the MAC of which is to be computed using key k_1 (assume both m and k_1 have a bit length equal to half that of h^s). The tag t would be (here I drop the s superscript to ease the notational burden):

$$t = h\Big\{2 \| h\big[m \| h(k_1 \| 0^n)\big]\Big\} \tag{4.2}$$

The reader might want to draw the diagrams corresponding to the previous and following calculations. Then we can compute $t' = h(3 \| t)$, and produce the forged message $m' = m \| 2$, for

[5] This description is a bit imprecise, but suffices for current purposes. For further details, see Definition 3.1 in Bellare, Canetti and Krawczyk [2].

[6] Also cf. §1.3 for important remarks on the importance of independent keys.

[7] Remember that for hash functions, s is a key, but not a secret one…

which the tag will be:

$$h\Big\{3\|h\Big\{2\|h\big[m\|h(k_1\|0^n)\big]\Big\}\Big\} = h(3\|t) = t' \tag{4.3}$$

The solution is to add another round of the Merkle-Damgård construction, but now using as inputs the output of the top row Merkle-Damgård round, and an independent key k_2—again, cf. figure 4.1, thinking of $k \oplus \mathsf{opad}$ as k_2. This can be abstracted as:

$$H^s\big(k_2\|H^s(k_1\|m)\big) \tag{4.4}$$

This is basically a strengthened version of the hash-and-MAC paradigm. This particular variant can be proven secure, if k_1 and k_2 are *independent*. But as we discussed above, using two keys for MACs is frowned upon. So what is done is to generate just one key, and XOR it with two values ipad and opad, fixed by the standard, and use those instead. Although now the keys are related—$k \oplus \mathsf{ipad} \oplus k \oplus \mathsf{opad}$ is always equal to the fixed value $\mathsf{ipad} \oplus \mathsf{opad}$, which would only happen with negligible probability if the keys were random—this new scheme can also be proved secure, albeit at the cost of imposing a stronger assumption on h^s—but this is out of the scope of the current course.[8]

4.5 Birthday Attacks

Consider the following problem: how many classmates do you need to have, for there to be a 50% chance of two of you sharing a birthday? This depends on the distribution of birthdays throughout the year, but for current purposes it suffices to assume a uniform distribution. Under this assumption, the answer might be surprising: on average, it suffices to have 23 students. This seems surprising because as there are at least 365 days in a year, one would intuitively suppose that with 23 students, the odds of a collision would be quite low. Which is not what happens, and this is why this result is sometimes nicknamed the "birthday paradox"—even though, strictly speaking, there is no paradox whatsoever here. A rigorous calculation is cumbersome—see §5.4.1 and §A.4 in [9]—but rounding 365 to 400, one notices that 23 is close to the square root of 400 (which is 20). And indeed, this is the intended take away of this discussion: for a set with n element, one needs on average $\sqrt{n}$ attempts to find a collision.

Why is this relevant to hash functions? Well, say you have an hash function H that produces an n-bit digest. Under a similar assumption as above, model H as a "random oracle", that is, assume that on a new input, H simply chooses a random element of the 2^n possible digests, according to a uniform distribution (but if queried on element for which a digest was previously chosen, then H outputs that same digest again). On average, the same reasoning applied above would entail that you need to evaluate H on $\sqrt{2^n} = 2^{n/2}$ inputs, before you can expect to find a collision. Notice that by the pigeonhole principle, after evaluating H on $2^n + 1$ different inputs, you find a collision with probability 1. The birthday bound significantly improves that result, with a probability that, while lower than 1, is still high enough to be realistic.

To get a feeling for this result, it might be instructive to compare it to the problem of inverting H on a specific output. I.e. say we have a digest y, and we are trying to find x such that $H(x) = y$. With one attempt, our probability of success is $1/2^n$. If we make another attempt, it succeeds with probability $1/(2^n - 1)$, but this second attempt only takes place if the first failed,

[8] For the curious: h^s needs to be such that the top row remains secure—i.e. indistinguishable from a pseudorandom function—under what is called a *related-key attack*. This can be shown to happen if h^s is a *Davies-Meyer* compression function ([9], §6.3.1)—which is what is used in for Merkle-Damgård constructions in practice.

which happens with probability $(2^n - 1)/2^n$. Hence the probability of succeeding after two attempts is:

$$\frac{1}{2^n} + \frac{2^n - 1}{2^n}\frac{1}{2^n - 1} = \frac{2}{2^n} \tag{4.5}$$

And so, after N attempts the probability of success is:

$$\frac{1}{2^n} + \prod_{i=2}^{N}\left(\frac{2^n - i}{2^n}\frac{1}{2^n - i}\right) = \frac{1}{2^n} + (N-1)\frac{1}{2^n} = \frac{N}{2^n} \tag{4.6}$$

Thus, to obtain a success probability of 50% in inverting H, we need $N = 2^n/2 = 2^{n-1}$ attempts. The reader should easily recognise that $2^{n-1} \gg 2^{n/2}$—in fact, this difference grows exponentially in n.[9] This is why inverting H (i.e. breaking pre-image resistance) is much harder than collision finding (breaking collision resistance)—and why the birthday bound should be known and understood by cryptography practitioners.[10]

4.6 Password Hashing and Key Derivation

The symmetric primitives we have seen all require keys that are generated *uniformly*. However, it is often more expedient for communicating parties to rely on some shared secret that will *not* be uniformly distributed. Just think of good old *passwords*: if they are directly used as a key, then the only keys that are used are those whose bits correspond to printable characters! This will be only a fraction of the entire keyspace—not what we want! To borrow from Katz and Lindell ([9], §5.6.4):

> In fact, the ASCII representations of the letters A–Z lie between 0x41 and 0x5A; in particular, the first 3 bits of every byte are always 010. This means that *37.5% of the bits of the resulting key will be fixed*, and the 128-bit key the parties derive will have only about 75 bits of entropy (i.e., there are only 2^{75} or so possibilities for the key).

The alternative is to use *key derivation functions*.[11] The reason hash functions play a role here, is that intuitively, they are very uncertain: it is very hard, looking at the input, to be able to guess anything about the tag. Technically, one says that the distribution of the digests has very high *entropy*—and in fact, the distribution with highest entropy is precisely the uniform distribution. But we do not this level of detail here; suffice to say that this uncertainty about what the digest of a given input will be means that we can use hash functions to take, say, passwords, and produce keys that more well-spread across the key space.

Password storage. It should be clear to anyone with even a modicum of background in computer or software engineering why storing passwords in the clear is a spectacularly dumb idea: should the database even be stolen, or leaked, even if partially, then the passwords can immediately be put to use; one just needs to read them. Worse, users may very well *reuse* their passwords, say across different sites. So a better strategy is required.

And a pretty simple one comes mind: instead of storing the plain password, run it through an hash function, and store the respective digest. And this is indeed a significant improvement, but

[9] This result also applies to other scenarios, e.g. brute force search: for a key of size n, the probability of finding a key after N attempts is $N/2^n$.

[10] As explained in §1.2, this implies that the security notion defined by the infeasibility of finding collisions is much *stronger* than the one defined by the infeasibility of finding a pre-image.

[11] Cf. `cryptography`'s documentation: Hazmat > Primitives > Key Derivation Functions.

it suffers from a significant problem. Namely the fact that users might use some easily guessable strings as passwords: the spouse's name, the dog's name, their favourite sports club, and so on. An attacker can *pre-compute* hashes for these easily guessable values, and he can do so months in advance. When and if he retrieves a password database, he just has to compare the values therein with the values he precomputed.

Hence we need a better strategy. One idea to use *slow hashing*, i.e. to deliberately make the password checking process slower. For example, by iterating the hash function several times (instead of using it just once). This will be barely noticeable by legitimate users, but might mean that months of pre-computation work now turn into years. Another approach is to use *salts*: we choose a random salt s for each user, and then in the password database store the pair $s, H(s||password)$. This means that when the attacker gets the database, he will the value of s, but will have to compute the hashes database only *after* he obtained the password database—and do so once for each user. This is adds a significant overhead to the attacker's work—and thus it is a widely used strategy to the attacker's work—and thus it is a widely used strategy.

5 RSA

Note: if the reader is unfamiliar with modular arithmetic and/or group theory, he could do worse than to check chapters 3 (groups and subgroups) and 4 (cyclic groups) of Judson [7]—going back to chapter 2 when needed (integers and divisibility). It is available online at http://abstract.ups.edu/.

5.1 The Chinese Remainder Theorem

Consider the following problem: given a family of integers $n_1, \ldots, n_k$, all pairwise relatively prime, and another family of integers $a_1, \ldots, a_k$, we want to find an integer a such that $a \equiv a_i \pmod{n_i}$, for all i. The way we do this is by finding a family of numbers $e_1, \ldots, e_k$, with the property that $e_i \equiv 1 \pmod{n_i}$, and $e_i \equiv 0 \pmod{n_j}$, for all $j \neq i$. Then it is straightforward to see that the number

$$a = \sum_i a_i e_i \tag{5.1}$$

solves the set of linear congruences. Indeed, modulo n_i we have:

$$a = a_i e_i + \sum_{j \neq i} a_j e_j \equiv a_i 1 + \sum_{j \neq i} a_j 0 = a_i \tag{5.2}$$

To construct the e_i, let $n = n_1 n_2 \ldots n_k$. Then $e_i = (n/n_i)(n/n_i)^{-1}$, where $(n/n_i)^{-1}$ denotes the modular inverse of n/n_i, the modulus being n_i.[1] Note this inverse always exists, as $gcd(n/n_i, n_i) = 1$, due to the fact that all the n_i are pairwise prime. Furthermore, for a given i, we have by construction that $e_i \equiv 1 \pmod{n_i}$—after all, e_i consists of a number multiplied by its (modular) inverse. And also by construction, $n_j \mid (n/n_i)$ for all $j \neq i$—and hence, $e_i \equiv 0 \pmod{n_j}$.

Now let $a' \equiv a \pmod{n}$. This means $n \mid (a - a')$, and as $n_i \mid n$, for all i, then also $n_i \mid (a - a')$, and thus $a' \equiv a \pmod{n_i}$. And finally, as we also have that $a \equiv a_i \pmod{n_i}$, we conclude that also $a' \equiv a_i \pmod{n_i}$, i.e. that a' is also a solution to the set of congruences.[2]

Conversely, suppose now that a' is a solution to the set of congruences, i.e. that $a' \equiv a_i \pmod{n_i}$ for all i. As we also have that $a \equiv a_i \pmod{n_i}$, we conclude that $a' \equiv a \pmod{n_i}$, again for all i. This equivalent to saying that $n_i \mid (a - a')$; and as this holds for all i, then we must also have that $lcm(n_1, \ldots, n_k) \mid (a - a')$. As $lcm(n_1, \ldots, n_k) = n$, due to the n_i being all pairwise relatively prime, we conclude that $n \mid (a - a')$, or equivalently, $a' \equiv a \pmod{n}$.

So to sum up, given a solution a to the set of congruences, any other integer a' is also a solution if and only if $a \equiv a' \pmod{n}$ (where $n = n_1 n_2 \ldots n_k$).

The CRT has a very neat interpretation in terms of residue classes [11, §2.5]. Indeed, suppose as above that a is a solution to the CRT congruences. Then we just saw that any other element

[1] **But note that you cannot reduce n/n_i modulo n_i!!** Otherwise it will no longer be a multiple of all the other n_j, with $j \neq i$!

[2] Equivalence relations, such as congruences ($\equiv$), are *transitive*: $a \equiv b$ and $b \equiv c$ implies $a \equiv c$.

of the residue class $[a]_n$ of $\mathbb{Z}_n$ is also a solution.[3] Furthermore, consider any one of the a_i; as $a \equiv a_i \pmod{n_i}$, we see that a belongs to the residue $[a_i]_{n_i}$, which thus coincides with the residue $[a]_{n_i}$. Hence the CRT be seen as a mapping from $\mathbb{Z}_n$ to $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$, as follows:

$$[a]_n \mapsto ([a]_{n_1} \times [a]_{n_2} \times \cdots \times [a]_{n_k}) \tag{5.3}$$

As any number congruent modulo n with a solution to the CRT is also a solution, we see that the mapping is well-defined (i.e., it does not depend on the particular element of the residue class). The mapping is also a bijection, which we can show as follows. First, I will show it is injective: if there are two equal tuples, say $\prod[a]_{n_i}$ and $\prod[b]_{n_i}$, then $a \equiv b \pmod{n_i}$, for all n_i. But as shown above, this implies that $a \equiv b \pmod{n}$, or equivalently, $[a]_n = [b]_n$. Hence the mapping is injective. For surjectiveness, the CRT algorithm itself, as outlined above, shows that given any tuple of elements in the $\mathbb{Z}_{n_i}$, there exists a corresponding element in $\mathbb{Z}_n$. Thus the mapping is surjective—and hence, bijective.

5.2 RSA: A Gentle Introduction

Pierre de Fermat's so-called little theorem tells us that for any integer a, and prime p, we have $a^p \equiv a \pmod{p}$. Now, if $a \not\equiv 0 \pmod{p}$, or equivalently, if a is not a multiple of p (symbolically, $p \nmid a$, read "p does not divide a"), the extended Euclidean Algorithm tells us that a has a modular inverse modulo p—indeed, it shows us how to compute it. That is, there exists an integer b such that $ab \equiv 1 \pmod{p}$. And so, if we multiply both sides of the first congruence above by b, we obtain $a^{p-1} \equiv 1 \pmod{p}$—valid for any a that is **not** a multiple of p.

Now, as your favourite book on abstract algebra or number theory will happily explain to you, when the modulus is not prime, things get a bit more complicated. In particular, we have to use *Euler's ϕ function*, also called the *totient* function. For a positive integer n, $\phi(n)$ equals the number of integers i such that $1 \leq i < n$ and $gcd(i, n) = 1$. This allows to generalise Fermat's little theorem as follows:

Theorem 5.1 (Euler's theorem). *If a and n are integers such that $n > 0$ and $gcd(a, n) = 1$, then the following holds:*

$$a^{\phi(n)} \equiv 1 \pmod{n} \tag{5.4}$$

Remark 5.2. If n is prime, then $\phi(n) = n - 1$ and we get back Fermat's (little) theorem. △

Most books explain RSA with the totient, but we can simplify things somewhat.[4] This is because the modulus used in RSA, though not a prime, has the (relatively) simple form of a product of two primes: $n = pq$. Now, just as above, we are trying to find a value t such that $x^t \equiv 1 \pmod{n}$, for almost all values of x. We already know we can set $t = \phi(n)$, in which case the condition holds for all x relatively prime to n. But we can do better. For $x^t \equiv 1 \pmod{n}$ means that $n \mid (x^t - 1)$, and as $n = pq$, this implies that $p \mid (x^t - 1)$ and $q \mid (x^t - 1)$—or equivalently, $x^t \equiv 1 \pmod{p}$ and $x^t \equiv 1 \pmod{q}$ respectively. Note that, as p and q are both primes, and hence also co-prime, we have $lcm(p, q) = pq = n$.[5] This means that any multiple of p and q is a multiple of n, and so the *converse* also holds: if $x^t \equiv 1 \pmod{p}$ and $x^t \equiv 1 \pmod{q}$, then also $x^t \equiv 1 \pmod{n}$.

[3] Recall that $[a]_n$ is composed of all integers b such that $b \equiv a \pmod{n}$, and that $\mathbb{Z}_n$ is the set of all such class, viz. $[0]_n, [1]_n, \ldots, [n-1]_n$.

[4] See Ferguson and Schneier ([5], §13).

[5] lcm stands for *least common multiple*; gcd for *greatest common divisor*.

So we just need to find t such that $x^t \equiv 1 \pmod{p}$ and $x^t \equiv 1 \pmod{q}$. Fermat's theorem comes to the rescue: to have $x^t \equiv 1 \pmod{p}$, we must have $(p-1) \mid t$. And similarly, to have $x^t \equiv 1 \pmod{q}$, we must have $(q-1) \mid t$. The smallest t for which this holds is $t = lcm(p-1,\ q-1)$, which you learned back in high school to compute as:

$$lcm(p-1,\ q-1) = \frac{(p-1)(q-1)}{gcd(p-1,\ q-1)} \tag{5.5}$$

So we set $t = lcm(p-1,\ q-1)$, for which $x^t \equiv 1 \pmod{n}$ holds—for *almost* all values of x. Which values of x should be excluded? Those for which either $x^t \equiv 1 \pmod{p}$ or $x^t \equiv 1 \pmod{q}$ fails to hold. Fermat's theorem says that these are the multiples of p and q, and as here we are working with modulo n, we want those multiples that are between 0 and $n-1$. So we have the multiples of p: $p, 2p, \ldots, (q-1)p$—so we have $q-1$ multiples of p. For q, we have $q, 2q, \ldots (p-1)q$—so $p-1$ multiples of q. And there is 0 (multiple of all numbers), so in total we have $(q-1)+(p-1)+1 = p+q-1$. For a large enough n, this is a very small fraction of the total number of values $0, \ldots, n-1$.

Remark 5.3. For $n = pq$, $\phi(n)$ is equal to the number of values from 1 to $n-1$ (so $pq-1$ values), minus the number of multiples of either p or q in that range. As we are excluding the 0, this is just $(q-1)+(p-1) = p+q-2$. And so we have $\phi(n) = (pq-1)-(p+q-2) = pq-p-q+1 = (p-1)(q-1)$. Comparing with (5.5), we see that $t \mid \phi(n)$, and so $x^{\phi(n)} \equiv 1 \pmod{n}$ also holds. So we have effectively proved theorem 5.1! △

RSA. This allows us to compute e, d such that $ed \equiv 1 \pmod{t}$, which means that ed can be written as $ed = tk+1$, for some value of k. Thus, if we encipher a message m as $m^e \pmod{n}$, and decipher the cryptogram doing $(m^e)^d \pmod{n}$, we obtain $m^{ed} = m^{tk+1} = (m^t)^k m \equiv m \pmod{n}$.[6]

This is the gist of the RSA asymmetric encryption algorithm. However, as presented, this algorithm—sometimes dubbed *textbook RSA*—is **extremely unsafe**. If nothing else, it is *deterministic*, which means it cannot even verify CPA security—let alone CCA. Addressing these shortcomings is the topic of section §5.3. Before that, however, I will add a clarifying remark about whether to use t as computed above, or $\phi(n)$.

Instead of computing $ed \equiv 1 \pmod{t}$, we could have computed $ed \equiv 1 \pmod{\phi(n)}$, and textbook RSA would still function: $m^{ed} = m^{\phi(n)k+1} = (m^{\phi(n)})^k m \equiv m \pmod{n}$. Moreover, the two versions are *interchangeable*: one can, e.g., compute $ed \equiv 1 \pmod{t}$, encipher with e, but decipher using a d' that verifies $ed' \equiv 1 \pmod{\phi(n)}$. This is because e has an inverse modulo t if and only if it has an inverse modulo $\phi(n)$. The backward direction is easy to see: e has an inverse modulo $\phi(n)$ if and only if $gcd(e, \phi(n)) = 1$, and from this latter condition comes (via the extended Euclidean algorithm) $er + \phi(n)s = 1$, for some integers r, s. But as was seen above, $t \mid \phi(n)$, which means that we can write the previous equality as $er + (tf)s = 1$, for some integer f. This means that $gcd(e, t) = 1$, and thus e also has an inverse modulo t. For the forward direction, we need the following lemma:

Lemma 5.4. *Let $n_1, \ldots, n_k$ be a family of integers, and let $n = \prod n_i$. Given an integer a, $gcd(a, n) = 1$ if and only if $gcd(a, n_i) = 1$, for $i = 1, \ldots, k$.*

Proof. See the appendix (§A.1).

[6] This assumes that m is already an element of $\mathbf{Z}_n$; see §5.3 for the problem of transforming an arbitrary message into a suitable integer.

Going back to RSA, if e has an inverse modulo t, then $gcd(e,t) = 1$. Now recall (5.5), and let $g = gcd(p-1, q-1)$. From above we have $t = lcm(p-1, q-1)$, and $\phi(n) = (p-1)(q-1)$. Hence we can write:

$$\phi(n) = t \times g \tag{5.6}$$

Notice t is a multiple of both $p-1$ and $q-1$, and g is a divisor of both $p-1$ and $q-1$—and hence, $g \mid t$. From $gcd(e,t) = 1$ comes $1 = er + ts$, for some integers r,s. This combined with the fact that t is a multiple of g immediately gives that $gcd(e,g) = 1$. We can now apply lemma 5.4 to conclude that $gcd(e, \phi(n)) = 1$, which entails that e has an inverse modulo $\phi(n)$. And we are done.

5.3 RSA made practical

tbd XXX

A | Preliminaries on Number Theory

A.1 Basics

One of the ways of the defining the *greatest common divisor* is straightforward: given two integers a and b, it is just the greatest of their non-negative common divisors. Given that 1 is a common divisor of every number, the set of non-negative common divisors is always nonempty, and it's also finite, *almost* always. The rub lies precisely in what happens when both numbers are 0: for then every integer is a common divisor, and thus there is no "greatest" common divisor. But we can work around that case.

Definition A.1. *Given two integers, not simultaneously zero, a and b, their* **greatest common divisor** *(gcd) is the greatest non-negative integer d such that it divides both a and b. If $a = b = 0$, then we define $gcd(0,0) = 0$.*

From this way of defining the gcd we get that $gcd(a,0) = gcd(0,a) = |a|$ holds for any integer a.

Theorem A.2. *Let a and b be two integers, and let $d = gcd(a,b)$. Then $d = xa + yb$, for some $x, y \in \mathbb{Z}$. Furthermore, every other common divisor of both a and b, also divides d.*

Proof. From the way we have defined the gcd, the result is obvious when either a or b, or both, are 0. So let us assume that neither is 0.

Consider the set $S = \{ r,s \in \mathbb{Z} \mid ar + bs \geq 1 \}$. This set is not empty (e.g. make $r = a$ and $s = b$); thus by the well-ordering principle, it contains a smallest element. Let $d = xa + by$ be that element. Dividing a by d, we get $a = dq+r \iff a = (xa+by)q+r \iff r = a(1-xq)-byq$. Thus the remainder is also a linear combination of a and b—which means that if $r > 0$, then $r \in S$. But r must be smaller than d, and d is, by assumption, supposed to be the smallest element of S—so r cannot belong to S. Hence we conclude that $r = 0$ (i.e. $d \mid a$). With b a similar reasoning shows that $d \mid b$—and thus d is a common divisor of both a and b.

Given that any number that divides a and b must divide any linear combination of theirs, we conclude that any common divisor of a and b must also divide d. In particular this also shows that d must be the *greatest* common divisor—indeed if d' were a common divisor that was greater than d, then we would have $d' \mid d$, which is a contradiction.[1] ■

Remark A.3. Given integers a, b, both their gcd d, as well as the integers x, y that allow us to write $d = ax + by$, are found via the *Extended Euclidean Algorithm.* △

Corollary A.4. *An integer r can be written as $r = as + bt$, if and only if $gcd(a,b) \mid r$.*

[1] In my view this already shows the gcd to be *unique*, for no set of integers can contain two distinct greatest elements. However, the gcd's uniqueness can also be shown explicitly: let d' now be another gcd. We would necessarily have $d \mid d'$ and $d' \mid d$, and as the gcd is always non-negative by definition, we conclude that $d = d'$.

Proof. As $gcd(a,b) = as + bt$ for some s,t, it is obvious that any multiple of the gcd can also be written as a linear combination of a and b. Conversely, any linear combination of a and b is divisible by any common divisor of a and b, and in particular by the gcd. ■

Theorem A.5. *If a and b are integers, then $gcd(a,b) = 1$ if and only if $xa + yb = 1$, for some integers x and y.*

Proof. If $gcd(a,b) = 1$, then by theorem A.2, $xa + yb = 1$, for some $x, y \in \mathbb{Z}$. Conversely if $xa + yb = 1$, then any common divisor of a and b must divide 1, which implies that the only non-negative common divisor of a and b is 1—and so $gcd(a,b) = 1$. ■

The case where the gcd of two integers is 1 is so important, it has its own name. It is of fundamental importance in algebra and number theory.

Definition A.6. *Two integers a, b such that $gcd(a,b) = 1$ are said to be* **relatively prime**.

The following result is needed in §5.2.

Lemma 5.4. *Let $n_1, \ldots, n_k$ be a family of integers, and let $n = \prod n_i$. Given an integer a, $gcd(a,n) = 1$ if and only if $gcd(a,n_i) = 1$, for $i = 1, \ldots, k$.*

Proof. ($\rightarrow$) If $gcd(a,n) = 1$, then $ax + ny = 1 \Leftrightarrow ax + \left(\prod n_i\right) y = 1$ from where we conclude that for each i we can write $ax + n_i y' = 1$, which entails that $gcd(a,n_i) = 1$.

($\leftarrow$) Let $ax_1 + n_1 y_1 = 1$ and $ax_2 + n_2 y_2 = 1$. Multiply one by the other; we obtain:

$$a^2 x_1 x_2 + a x_1 n_2 y_2 + n_1 y_1 a x_2 + n_1 y_1 n_2 y_2 = ax' + n_1 n_2 y' = 1^2 = 1 \tag{A.1}$$

If we now have that $ax_3 + n_3 y_3 = 1$, then multiplying member-wise by $ax' + n_1 n_2 y' = 1$ will yield $ax'' + n_1 n_2 n_3 y'' = 1$. Now suppose that we have shown that $ar + (n_1 n_2 \ldots n_j)s = 1$, for numbers $n_1, n_2, \ldots$, up to n_j (for some integers r,s). Now as $gcd(a, n_{j+1}) = 1$, there are r', s' such that $ar' + n_{j+1}s' = 1$. Doing sidewise multiplication, as in (A.1), will now yield $ar'' + (n_1 n_2 \ldots n_j n_{j+1})s'' = 1$, for some r'', s'', which, by induction, shows the result. ■

A.2 Modular Arithmetic

An integer a has an inverse modulo n if and only if $gcd(a,n) = 1$. Indeed, if $gcd(a,n) = 1$, then we can write $ax + ny = 1$ (cf. theorem A.2)—and so, x is the modular inverse of a, modulo n. Conversely, if a has an inverse modulo n, say x, then this means that $ax \equiv 1 \pmod{n}$, or equivalently, $ax = 1 + kn$, for some integer k. Rewrite this as $ax + (-k)n = 1$, and from theorem A.5 we conclude that $gcd(a,n) = 1$.

References

1. **Arora**, Sanjeev and Boaz **Barak** (2009). *Complexity Theory: A Modern Approach*. Cambridge, UK: Cambridge University Press. ISBN: 978-0-521-42426-4. Cited on page 6.
2. **Bellare**, Mihir, Ran **Canetti**, and Hugo **Krawczyk** (1996). *Keying Hash Functions for Message Authentication*. In Koblitz, N. (ed.), Advances in Cryptology — CRYPTO '96. Berlin, Heidelberg: Springer, pp. 1–15. Cited on page 31.
3. **Bellovin**, Steve (2009). Email sent to `comp.risks` (Risks Digest), 25.71, June 6, 2009. It used to be availabe at `https://groups.google.com/forum/#!topic/comp.risks/4V3cECtN_vQ` (last checked at 2016-05-09), but that URL is no longer accessible. Cited on page 3.
4. **Benoit**, Viguier (2016). `https://gitlab.insa-rennes.fr/bviguier/mri/blob/fd9d822a6cb4eea2bce422701a2b011811d78cb1/MRI/tikz/CBC-MAC.tex` Cited on page 22.
5. **Ferguson**, Niels and Bruce **Schneier** (2003). *Practical Cryptography*. Indianapolis: Wiley Publishing. ISBN: 0-471-22357-3. Cited on pages 3, 4, and 36.
6. **Jean**, Jérémy (2016). *TikZ for Cryptographers*. URL: `https://www.iacr.org/authors/tikz/`. Cited on page 41.
7. **Judson**, Thomas W. (2018). *Abstract Algebra*. Ann Arbor, Michigan: Orthogonal Publishing. ISBN: 978-1-944325-8. Cited on page 35.
8. **Katz**, Jonathan and Moti **Yung** (2001). *Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation*. In Goos, G., Hartmanis, J., van Leeuwen, J., and Schneier, B. (eds.), Fast Software Encryption. Berlin, Heidelberg: Springer, pp. 284–299. Cited on page 23.
9. **Katz**, Jonathan and Yehuda **Lindell** (2015). *Introduction to Modern Cryptography*, 2nd edition. Boca Raton, FL, U.S.: CRC Press. ISBN: 978-1-4665-7027-6. Cited on pages 5, 6, 8, 9, 10, 11, 12, 13, 19, 20, 21, 23, 30, 31, 32, and 33.
10. **Maimut**, Diana (2017). See [6]. Cited on page 16.
11. **Shoup**, Victor (2008). *A Computational Introduction to Number Theory and Algebra*, 2nd edition. New York: Cambridge University Press. Cited on page 35.