# Crypto 101

**lvh**

*Pomidorkowi*

# Contents

# Part I

# Foreword

1

# About this book

> Lots of people working in cryptography have no deep concern with real application issues. They are trying to discover things clever enough to write papers about.
>
> *Whitfield Diffie*

This book is intended as an introduction to cryptography for programmers of any skill level. It's a continuation of a talk of the same name, which was given by the author at PyCon 2013.

The structure of this book is very similar: it starts with very simple primitives, and gradually introduces new ones, demonstrating why they're necessary. Eventually, all of this is put together into complete, practical cryptosystems, such as TLS, GPG and *OTR*.

The goal of this book is not to make anyone a cryptographer or a security researcher. The goal of this book is to understand how complete cryptosystems work from a bird's eye view, and how to apply them in real software.

The exercises accompanying this book focus on teaching cryptography by breaking inferior systems. That way, you won't just "know" that some particular thing is broken; you'll know exactly *how* it's broken, and that you, yourself, armed with little more than some spare time and your favorite programming language, can break them. By seeing how these ostensibly secure systems are actually completely broken, you will understand *why* all these primitives and constructions are necessary for complete cryptosystems. Hopefully, these exercises will also leave you with healthy distrust of DIY cryptography in all its forms.

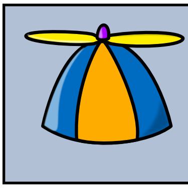
For a long time, cryptography has been deemed the exclusive realm of experts. From the many internal leaks we've seen over the years of the internals of both large and small corporations alike, it has become obvious that that approach is doing more harm than good. We can no longer afford to keep the two worlds strictly separate. We must join them into one world where all programmers are educated in the basic underpinnings of information security, so that they can work together with information security professionals to produce more secure software systems for everyone. That does not make people such as penetration testers and security researchers obsolete or less valuable; quite the opposite, in fact. By sensitizing all programmers to security concerns, the need for professional security audits will become more apparent, not less.

This book hopes to be a bridge: to teach everyday programmers from any field or specialization to understand just enough cryptography to do their jobs, or maybe just satisfy their appetite.

# 2

# Advanced sections

This book is intended as a practical guide to cryptography for programmers. Some sections go into more depth than they need to in order to achieve that goal. They're in the book anyway, just in case you're curious; but I generally recommend skipping these sections. They'll be marked like this:



> This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

3

# Development

The entire Crypto 101 project is publicly developed on GitHub under the `crypto101` organization, including this book.

This is an early pre-release of this book. All of your questions, comments and bug reports are highly appreciated. If you don't understand something after reading it, or a sentence is particularly clumsily worded, *that's a bug* and I would very much like to fix it! Of course, if I never hear about your issue, it's very hard for me to address...

The copy of this book that you are reading right now is based on the git commit with hash 64e8ccf, also known as 0.6.0-95-g64e8ccf.

4

# Acknowledgments

This book would not have been possible without the support and contributions of many people, even before the first public release. Some people reviewed the text, some people provided technical review, and some people helped with the original talk. In no particular order:

- My wife, Ewa
- Brian Warner
- Oskar Żabik
- Ian Cordasco
- Zooko Wilcox-O'Hearn
- Nathan Nguyen (@nathanhere)

Following the public release, many more people contributed changes. I'd like to thank the following people in particular (again, in no particular order):

- coh2, for work on illustrations

- TinnedTuna, for review work on the XOR section (and others)
- dfc, for work on typography and alternative formats
- jvasile, for work on typefaces and automated builds
- hmmueller, for many, many notes and suggestions
- postboy (Ivan Zuboff), for many reported issues
- EdOverflow, for many contributions
- gliptak (Gábor Lipták) for work on automating builds,

as well as the huge number of people that contributed spelling, grammar and content improvements. Thank you!

# Part II

# Building blocks

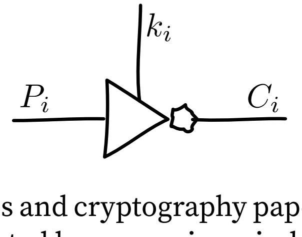
# 5

# Exclusive or

## 5.1 Description

Exclusive or, often called “XOR”, is a Boolean[1] binary[2] operator that is true when either the first input or the second input, but not both, are true.

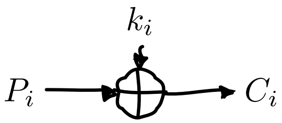
Another way to think of XOR is as something called a “programmable inverter”: one input bit decides whether to invert the other input bit, or to just pass it through unchanged. “Inverting” bits is colloquially called “flipping” bits, a term we’ll use often throughout the book.



In mathematics and cryptography papers, exclusive or is generally represented by a cross in a circle: $\oplus$. We’ll use the same notation in this book:

[1] Uses only “true” and “false” as input and output values.

[2] Takes two parameters.



The inputs and output here are named as if we're using XOR as an encryption operation. On the left, we have the plaintext bit $P_i$. The $i$ is just an index, since we'll usually deal with more than one such bit. On top, we have the key bit $k_i$, that decides whether or not to invert $P_i$. On the right, we have the ciphertext bit, $C_i$, which is the result of the XOR operation.

## 5.2 A few properties of XOR

Since we'll be dealing with XOR extensively during this book, we'll take a closer look at some of its properties. If you're already familiar with how XOR works, feel free to skip this section.

We saw that the output of XOR is 1 when one input or the other (but not both) is 1:

$$\begin{array}{cc} 0 \oplus 0 = 0 & 1 \oplus 0 = 1 \\ 0 \oplus 1 = 1 & 1 \oplus 1 = 0 \end{array}$$

There are a few useful arithmetic tricks we can derive from that.

1. You can apply XOR in any order: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$

2. You can flip the operands around: $a \oplus b = b \oplus a$

3. Any bit XOR itself is 0: $a \oplus a = 0$. If $a$ is 0, then it's $0 \oplus 0 = 0$; if $a$ is 1, then it's $1 \oplus 1 = 0$.

4. Any bit XOR 0 is that bit again: $a \oplus 0 = a$. If $a$ is 0, then it's $0 \oplus 0 = 0$; if $a$ is 1, then it's $1 \oplus 0 = 1$.

These rules also imply $a \oplus b \oplus a = b$:

$$\begin{aligned} a \oplus b \oplus a &= a \oplus a \oplus b && \text{(second rule)} \\ &= 0 \oplus b && \text{(third rule)} \\ &= b && \text{(fourth rule)} \end{aligned}$$

We'll use this property often when using XOR for encryption; you can think of that first XOR with $a$ as encrypting, and the second one as decrypting.

## 5.3 Bitwise XOR

XOR, as we've just defined it, operates only on single bits or Boolean values. Since we usually deal with values comprised of many bits, most programming languages provide a "bitwise XOR" operator: an operator that performs XOR on the respective bits in a value.

Python, for example, provides the ^ (caret) operator that performs bitwise XOR on integers. It does this by first expressing those two integers in binary[3], and then performing XOR on their respective bits. Hence the name, *bitwise* XOR.

$$\begin{aligned} 73 \oplus 87 &= 0b1001001 \oplus 0b1010111 \\ &= \begin{array}{cccccccl} 1 & 0 & 0 & 1 & 0 & 0 & 1 & \quad\text{(left)} \\ \oplus & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus & \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & \quad\text{(right)} \end{array} \\ &= \begin{array}{ccccccc} 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array} \\ &= 0b0011110 \\ &= 30 \end{aligned}$$

## 5.4 One-time pads

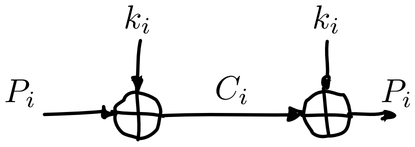
XOR may seem like an awfully simple, even trivial operator. Even so, there's an encryption scheme, called a one-time

[3] Usually, numbers are already stored in binary internally, so this doesn't actually take any work. When you see a number prefixed with "0b", the remaining digits are a binary representation.

pad, which consists of just that single operator. It's called a one-time pad because it involves a sequence (the "pad") of random bits, and the security of the scheme depends on only using that pad once. The sequence is called a pad because it was originally recorded on a physical, paper pad.

This scheme is unique not only in its simplicity, but also because it has the strongest possible security guarantee. If the bits are truly random (and therefore unpredictable by an attacker), and the pad is only used once, the attacker learns nothing about the plaintext when they see a ciphertext.[4]

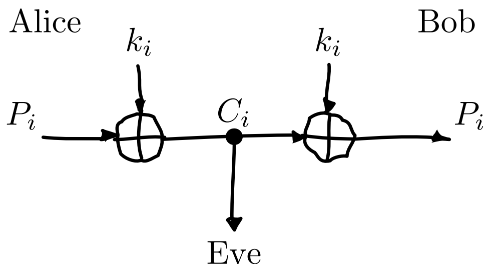
Suppose we can translate our plaintext into a sequence of bits. We also have the pad of random bits, shared between the sender and the (one or more) recipients. We can compute the ciphertext by taking the bitwise XOR of the two sequences of bits.



If an attacker sees the ciphertext, we can prove that they will learn zero information about the plaintext without the key. This property is called *perfect security*. The proof can be understood intuitively by thinking of XOR as a programmable inverter, and then looking at a particular bit intercepted by Eve, the eavesdropper.

Let's say Eve sees that a particular ciphertext bit $c_i$ is 1. She has no idea if the matching plaintext bit $p_i$ was 0 or 1, because she has no idea if the key bit $k_i$ was 0 or 1. Since all of the key bits are truly random, both options are exactly equally probable.

[4] The attacker does learn that the message exists, and, in this simple scheme, the length of the message. While this typically isn't too important, there are situations where this might matter, and there are secure cryptosystems to both hide the existence and the length of a message.



## 5.5 Attacks on "one-time pads"

The one-time pad security guarantee only holds if it is used correctly. First of all, the one-time pad has to consist of truly random data. Secondly, the one-time pad can only be used once (hence the name). Unfortunately, most commercial products that claim to be "one-time pads" are snake oil[5], and don't satisfy at least one of those two properties.

### Not using truly random data

The first issue is that they use various deterministic constructs to produce the one-time pad, instead of using truly random data. That isn't necessarily insecure: in fact, the most obvious example, a synchronous stream cipher, is something we'll see later in the book. However, it does invalidate the "unbreakable" security property of one-time pads. The end user would be better served by a more honest cryptosystem, instead of one that lies about its security properties.

### Reusing the "one-time" pad

The other issue is with key reuse, which is much more serious. Suppose an attacker gets two ciphertexts with the same "one-time" pad. The attacker can then XOR the two cipher-

[5] "Snake oil" is a term for all sorts of dubious products that claim extraordinary benefits and features, but don't really realize any of them.

texts, which is also the XOR of the plaintexts:

$$\begin{aligned}
c_1 \oplus c_2 &= (p_1 \oplus k) \oplus (p_2 \oplus k) && \text{(definition)} \\
&= p_1 \oplus k \oplus p_2 \oplus k && \text{(reorder terms)} \\
&= p_1 \oplus p_2 \oplus k \oplus k && (a \oplus b = b \oplus a) \\
&= p_1 \oplus p_2 \oplus 0 && (x \oplus x = 0) \\
&= p_1 \oplus p_2 && (x \oplus 0 = x)
\end{aligned}$$

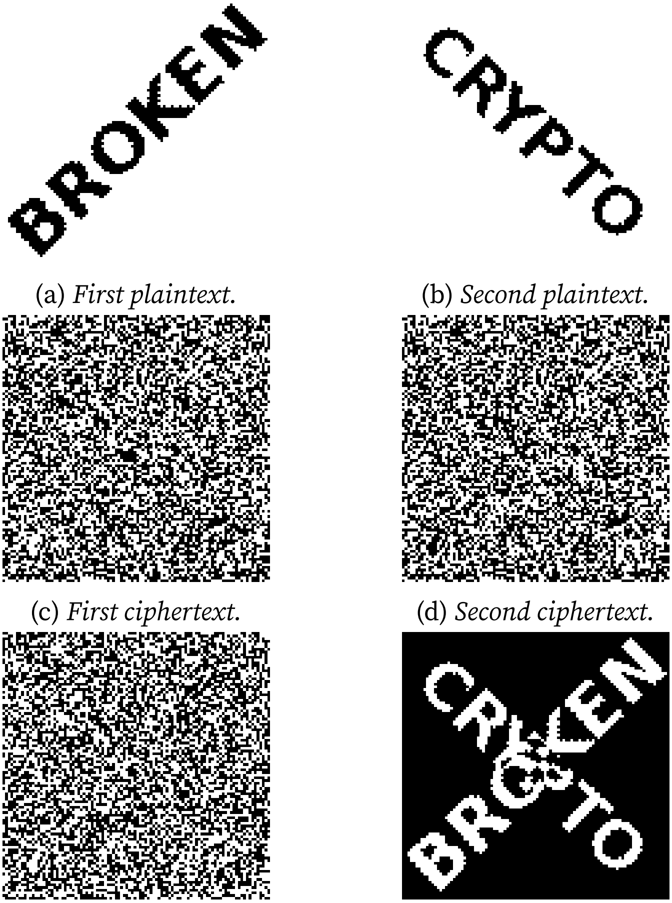
At first sight, that may not seem like an issue. To extract either $p_1$ or $p_2$, you'd need to cancel out the XOR operation, which means you need to know the other plaintext. The problem is that even the result of the XOR operation on two plaintexts contains quite a bit information about the plaintexts themselves. We'll illustrate this visually with some images from a broken "one-time" pad process, starting with Figure 5.1.

## Crib-dragging

A classical approach to breaking multi-time pad systems involves "crib-dragging", a process that uses small sequences that are expected to occur with high probability. Those sequences are called "cribs". The name crib-dragging originated from the fact that these small "cribs" are dragged from left to right across each ciphertext, and from top to bottom across the ciphertexts, in the hope of finding a match somewhere. Those matches form the sites of the start, or "crib", if you will, of further decryption.

The idea is fairly simple. Suppose we have several encrypted messages $C_i$ encrypted with the same "one-time" pad $K$[6]. If we could correctly guess the plaintext for one of

[6] We use capital letters when referring to an entire message, as opposed to just bits of a message.



(a) *First plaintext.*

(b) *Second plaintext.*

(c) *First ciphertext.*

(d) *Second ciphertext.*

(e) *Reused key.*

(f) *XOR of ciphertexts.*

Figure 5.1: Two plaintexts, the re-used key, their respective ciphertexts, and the XOR of the ciphertexts. Information about the plaintexts clearly leaks through when we XOR the ciphertexts.

the messages, let's say $C_j$, we'd know $K$:

$$\begin{aligned} C_j \oplus P_j &= (P_j \oplus K) \oplus P_j \\ &= K \oplus P_j \oplus P_j \\ &= K \oplus 0 \\ &= K \end{aligned}$$

Since $K$ is the shared secret, we can now use it to decrypt all of the other messages, just as if we were the recipient:

$$P_i = C_i \oplus K \qquad \text{for all } i$$

Since we usually can't guess an entire message, this doesn't actually work. However, we might be able to guess parts of a message.

If we guess a few plaintext bits $p_i$ correctly for *any* of the messages, that would reveal the key bits at that position for *all* of the messages, since $k = c_i \oplus p_i$. Hence, all of the plaintext bits at that position are revealed: using that value for $k$, we can compute the plaintext bits $p_i = c_i \oplus k$ for all the other messages.

Guessing parts of the plaintext is a lot easier than guessing the entire plaintext. Suppose we know that the plaintext is in English. There are some sequences that we know will occur very commonly, for example (the ␣ symbol denotes a space):

- `␣the␣` and variants such as `.␣The␣`
- `␣of␣` and variants
- `␣to␣` and variants
- `␣and␣` (no variants; only occurs in the middle of a sentence)
- `␣a␣` and variants

If we know more about the plaintext, we can make even better guesses. For example, if it's HTTP serving HTML, we would expect to see things like `Content-Type`, `<a>`, and so on.

That only tells us which plaintext sequences are likely, giving us likely guesses. How do we tell if any of those guesses are correct? If our guess is correct, we know all the other plaintexts at that position as well, using the technique described earlier. We could simply look at those plaintexts and decide if they look correct.

In practice, this process needs to be automated because there are so many possible guesses. Fortunately that's quite easy to do. For example, a very simple but effective method is to count how often different symbols occur in the guessed plaintexts: if the messages contain English text, we'd expect to see a lot of letters e, t, a, o, i, n. If we're seeing binary nonsense instead, we know that the guess was probably incorrect, or perhaps that message is actually binary data.

These small, highly probable sequences are called "cribs" because they're the start of a larger decryption process. Suppose your crib, `the`, was successful and found the five-letter sequence `t thr` in another message. You can then use a dictionary to find common words starting with `thr`, such as `through`. If that guess were correct, it would reveal four more bytes in all of the ciphertexts, which can be used to reveal even more. Similarly, you can use the dictionary to find words ending in `t`.

This becomes even more effective for some plaintexts that we know more about. If some HTTP data has the plaintext `ent-Len` in it, then we can expand that to `Content-Length:`, revealing many more bytes.

While this technique works as soon as two messages are encrypted with the same key, it's clear that this becomes even easier with more ciphertexts using the same key, since all of the steps become more effective:

- We get more cribbing positions.

- More plaintext bytes are revealed with each successful crib and guess, leading to more guessing options elsewhere.
- More ciphertexts are available for any given position, making guess validation easier and sometimes more accurate.

These are just simple ideas for breaking multi-time pads. While they're already quite effective, people have invented even more effective methods by applying advanced, statistical models based on natural language analysis. This only demonstrates further just how broken multi-time pads are. [MWES06]

## 5.6 Remaining problems

Real one-time pads, implemented properly, have an extremely strong security guarantee. It would appear, then, that cryptography is over: encryption is a solved problem, and we can all go home. Obviously, that's not the case.

One-time pads are rarely used, because they are horribly impractical: the key is at least as large as all information you'd like to transmit, *put together*. Plus, you'd have to exchange those keys securely, ahead of time, with all people you'd like to communicate with. We'd like to communicate securely with everyone on the Internet, and that's a very large number of people. Furthermore, since the keys have to consist of truly random data for its security property to hold, key generation is fairly difficult and time-consuming without specialized hardware.

One-time pads pose a trade-off. It's an algorithm with a solid information-theoretic security guarantee, which you can not get from any other system. On the other hand, it also has extremely impractical key exchange requirements. However, as we'll see throughout this book, secure symmetric encryption algorithms aren't the pain point of modern cryptosystems. Cryptographers have designed plenty of those, while practical key management remains one of the toughest challenges facing modern cryptography. One-time pads may solve a problem, but it's the wrong problem.

While they may have their uses, they're obviously not a panacea. We need something with manageable key sizes while maintaining secrecy. We need ways to negotiate keys over the Internet with people we've never met before.

6

# Block ciphers

> Few false ideas have more firmly gripped the minds of so many intelligent men than the one that, if they just tried, they could invent a cipher that no one could break.
>
> *David Kahn*

## 6.1 Description

A *block cipher* is an algorithm that encrypts blocks of a fixed length. The encryption function $E$ transforms plaintext blocks $P$ into ciphertext blocks $C$ by using a secret key $k$:

$$C = E(k, P)$$

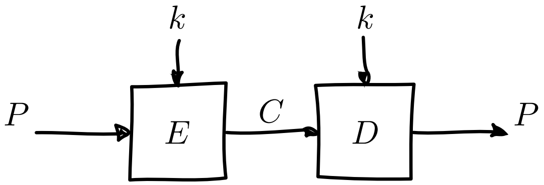
Plaintext and ciphertext blocks are sequences of bits and always match in size. The block cipher's *block size* is a fixed size. *Keyspace* is the set of all possible keys.

Once we encrypt plaintext blocks into ciphertext blocks, they are later decrypted to recover original plaintext block.

The original plaintext block $P$ is produced using a decryption function $D$. It takes the ciphertext block $C$ and the key $k$ (the same one used to encrypt the block) as inputs.

$$P = D(k, C)$$

Or, visually represented in blocks:



A block cipher is an example of a *symmetric-key encryption* scheme, also known as a *secret-key encryption* scheme. The same secret key is used for both encryption and decryption. Later in the book, we contrast this with *public-key encryption* algorithms, which have a distinct key for encryption and decryption.

A block cipher is a *keyed permutation*. It is a *permutation* because the block cipher maps each possible block to another block. It is also a *keyed* permutation because the key determines exactly which blocks map to which. It is important for the block cipher to be a permutation because the recipient must map blocks back to the original blocks.

We illustrate this by looking at a block cipher with an impractical, tiny 4-bit block size. $2^4 = 16$ possible blocks. Since each of the blocks map to a hexadecimal digit, we represent the blocks by that digit. Figure 6.1 illustrates blocks that the cipher operates on.

Once we select a secret key, the block cipher uses it to determine the encryption of any given block. We illustrate that relationship with an arrow. The tail of the arrow has the block encrypted with $E$ under key $k$ and the arrowhead is mapped to the block.

In Figure 6.2, note that the permutation is not just one big cycle. It contains a large cycle of 7 elements, and several

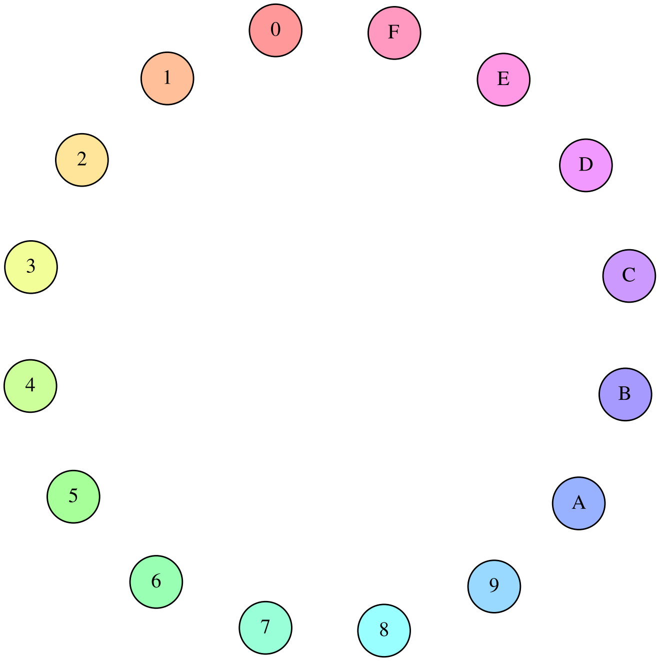


Figure 6.1: All 16 nodes operated on by the block cipher. Each node is designated by a hexadecimal digit.

smaller cycles of 4, 3 and 2 elements each. It is also perfectly possible that an element encrypts to itself. This is to be expected when selecting random permutations, which is approximately what a block cipher is doing; it doesn't demonstrate a bug in the block cipher.

When you decrypt instead of encrypt, the block cipher computes the inverse permutation. In Figure 6.3, we get the same illustration. The difference between the illustrations is that all arrowheads point in the opposite direction.

The key defines which blocks map to which blocks. A different key would lead to a different set of arrows, as you

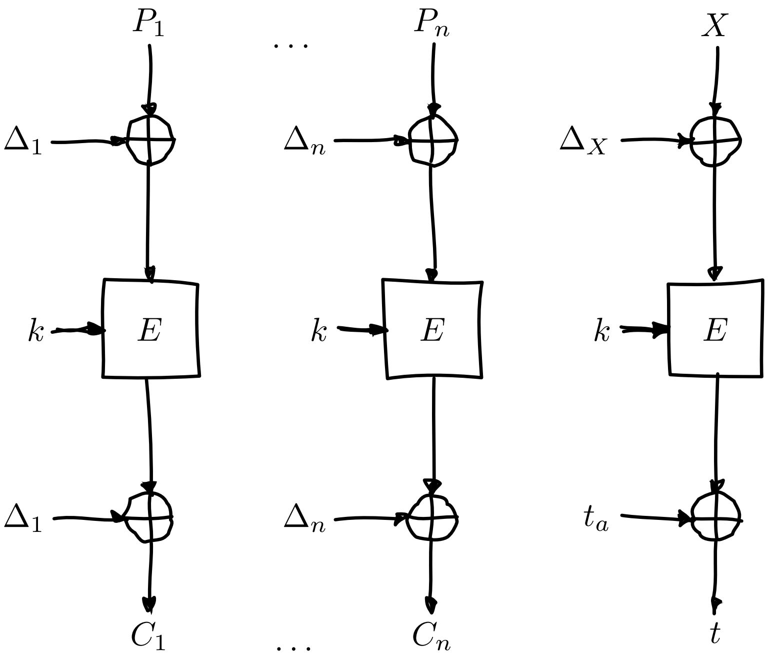


Figure 6.2: An encryption permutation made by a block cipher under a particular key $k$.

can see in Figure 6.4.

In this illustration, you'll even notice that there are two permutations of length 1: an element that maps to itself. This is again something to be expected when selecting random permutations.

Knowing a bunch of (input, output) pairs for a given key shouldn't give you any information about any other (input, output) pairs under that key[1]. As long as we're talking about a hypothetical perfect block cipher, there's no easier way to decrypt a block other than to "brute-force" the key: i.e. just try every single one of them until you find the right one.

[1] The attentive reader may have noticed that this breaks in the extremes: if you know all but one of the pairs, then you know the last one by exclusion.

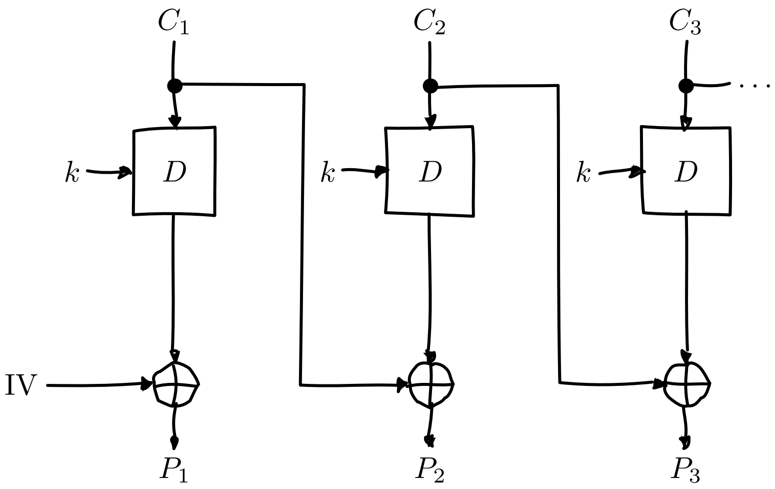


Figure 6.3: The decryption permutation produced by the block cipher under the same key $k$. It is the inverse of the encryption permutation in that all arrowheads reverse.

Our toy illustration block cipher only has 4 bit blocks, or $2^4 = 16$ possibilities. Real, modern block ciphers have much larger block sizes, such as 128 bits, or $2^{128}$ (slightly more than $10^{38.5}$) possible blocks. Mathematics tells us that there are $n!$ (pronounced "$n$ factorial") different permutations of an $n$ element set. It's defined as the product of all of the numbers from 1 up to and including $n$:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Factorials grow incredibly quickly. For example, $5! = 120$, $10! = 3628800$, and the rate continues to increase. The number of permutations of the set of blocks of a cipher with a 128 bit block size is $(2^{128})!$. Just $2^{128}$ is large already (it takes 39 digits to write it down), so $(2^{128})!$ is a mind-bogglingly huge number, impossible to comprehend. Common key sizes are only in the range of 128 to 256 bits, so there are only between $2^{128}$ and $2^{256}$ permutations a cipher can perform. That's just a tiny fraction of all possible permutations of the blocks,

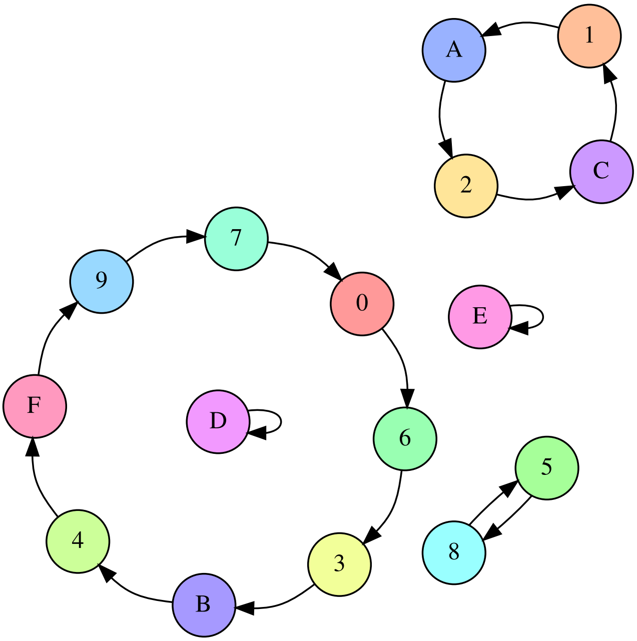


Figure 6.4: An encryption permutation produced by the block cipher under a different key.

but that's okay: that tiny fraction is still nowhere near small enough for an attacker to just try them all.

Of course, a block cipher should be as easy to compute as possible, as long as it doesn't sacrifice any of the above properties.

## 6.2 AES

The most common block cipher in current use is AES.

Contrary to its predecessor DES (which we'll look at in more detail in the next chapter), AES was selected through a public, peer-reviewed competition following an open call for proposals. This competition involved several rounds where all of the contestants were presented, subject to extensive cryptanalysis, and voted upon. The AES process was well-received among cryptographers, and similar processes are generally considered to be the preferred way to select cryptographic standards.

Prior to being chosen as the Advanced Encryption Standard, the algorithm was known as Rijndael, a name derived from the two last names of the Belgian cryptographers that designed it: Vincent Rijmen and Joan Daemen. The Rijndael algorithm defined a family of block ciphers, with block sizes and key sizes that could be any multiple of 32 bits between 128 bits and 256 bits. [DR02] When Rijndael became AES through the FIPS standardization process, the parameters were restricted to a block size of 128 bits and keys sizes of 128, 192 and 256 bits. [fip01]

There are no practical attacks known against AES. While there have been some developments in the last few years, most of them involve related-key attacks [BK09], some of them only on reduced-round versions of AES [BDK+09].[2]

[2] Symmetric algorithms usually rely on a round function to be repeated a number of times. Typically each invocation involves a "round key" derived from the main key. A reduced-round version is intentionally easier to attack. These attacks can give insight as to how resistant the full cipher is.

A related key attack involves making some predictions about how AES will behave under several different keys with some specific mathematical relation. These relations are fairly simple, such as XORing with an attacker-chosen constant. If an attacker is allowed to encrypt and decrypt a large number of blocks with these related keys, they can attempt to recover the original key with significantly less computation than would ordinarily be necessary to crack it.

While a theoretically ideal block cipher wouldn't be vulnerable to a related key attack, these attacks aren't considered practical concerns. In practice cryptographic keys are generated via a cryptographically secure pseudorandom number generator, or a similarly secure *key agreement* scheme or key derivation scheme (we'll see more about those later). Therefore, the odds of selecting two such related keys by accident is nonexistent. These attacks are interesting from an academic perspec-

## A closer look at Rijndael

> This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

AES consists of several independent steps. At a high level, AES is a *substitution-permutation network*.

### Key schedule

AES requires separate keys for each round in the next steps. The key schedule is the process which AES uses to derive 128-bit keys for each round from one master key.

First, the key is separated into 4 byte columns. The key is rotated and then each byte is run through an S-box (substitution box) that maps it to something else. Each column is then XORed with a round constant. The last step is to XOR the result with the previous round key.

The other columns are then XORed with the previous round key to produce the remaining columns.

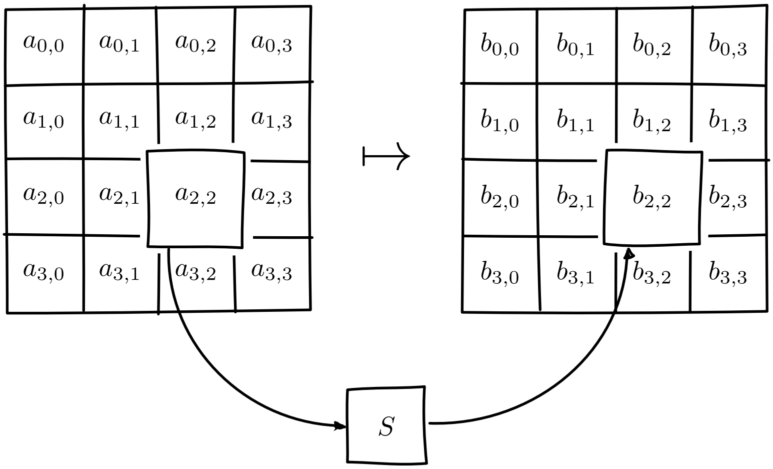
### SubBytes

SubBytes is the step that applies the S-box (substitution box) in AES. The S-box itself substitutes a byte with another byte, and this S-box is applied to each byte in the AES state.

It works by taking the multiplicative inverse over the Galois field, and then applying an affine transformation so that there are no values $x$ so that $x \oplus S(x) = 0$ or $x \oplus S(x) = \texttt{0xff}$. To rephrase: there are no values of $x$ that the substitution box maps to $x$ itself, or $x$ with all bits flipped. This makes

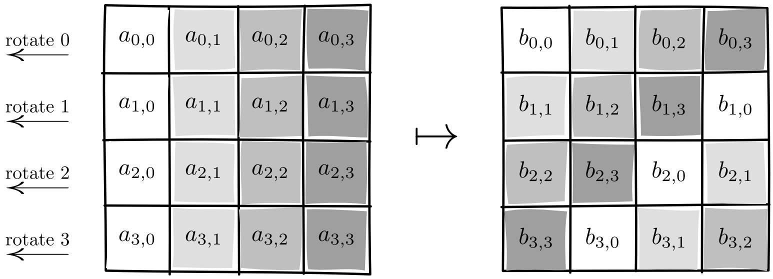
tive: they can help provide insight in the workings of the cipher, guiding cryptographers in designing future ciphers and attacks against current ciphers.

the cipher resistant to linear cryptanalysis, unlike the earlier DES algorithm, whose fifth S-box caused serious security problems.[3]


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$\mapsto$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$S$

## ShiftRows

After having applied the SubBytes step to the 16 bytes of the block, AES shifts the rows in the $4 \times 4$ array:

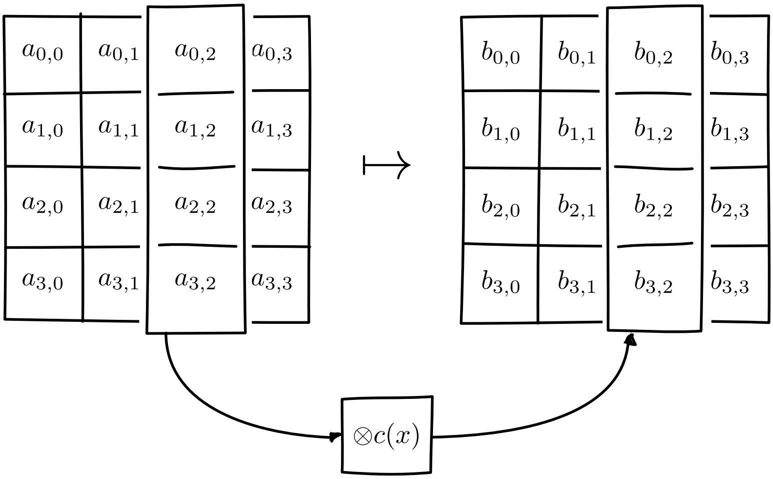

rotate 0
rotate 1
rotate 2
rotate 3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$\mapsto$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{1,0}$
$b_{2,2}$ $b_{2,3}$ $b_{2,0}$ $b_{2,1}$
$b_{3,3}$ $b_{3,0}$ $b_{3,1}$ $b_{3,2}$

[3] In its defense, linear attacks were not publicly known back when DES was designed.

### MixColumns

MixColumns multiplies each column of the state with a fixed polynomial.

ShiftRows and MixColumns represent the diffusion properties of AES.

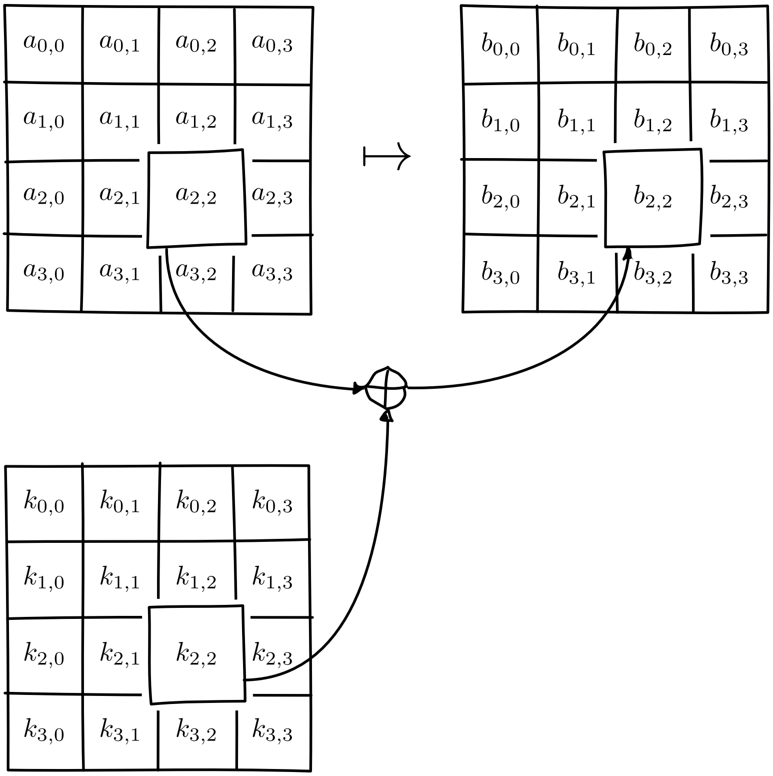

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$\mapsto$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$\otimes c(x)$

### AddRoundKey

As the name implies, the AddRoundKey step adds the bytes from the round key produced by the key schedule to the state of the cipher.

## 6.3 DES and 3DES

The DES is one of the oldest block ciphers that saw widespread use. It was published as an official FIPS standard in 1977. It is no longer considered secure, mainly due to its tiny key size of 56 bits. (The DES algorithm actually takes a 64 bit key input, but the remaining 8 bits are only used for parity checking, and are discarded immediately.) It shouldn't be used in new systems. On modern hardware, DES can be brute forced in less than a day. [Gmb08]


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$\mapsto$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$\oplus$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$

In an effort to extend the life of the DES algorithm, in a way that allowed much of the spent hardware development effort to be reused, people came up with 3DES: a scheme where input is first encrypted, then decrypted, then encrypted again:

$$C = E_{DES}(k_1, D_{DES}(k_2, E_{DES}(k_3, p)))$$

This scheme provides two improvements:

- By applying the algorithm three times, the cipher becomes harder to attack directly through cryptanalysis.
- By having the option of using many more total key bits, spread over the three keys, the set of all possible keys

becomes much larger, making brute-forcing impractical.

The three keys could all be chosen independently (yielding 168 key bits), or $k_3 = k_1$ (yielding 112 key bits), or $k_1 = k_2 = k_3$, which, of course, is just plain old DES (with 56 key bits). In the last keying option, the middle decryption reverses the first encryption, so you really only get the effect of the last encryption. This is intended as a backwards compatibility mode for existing DES systems. If 3DES had been defined as $E(k_1, E(k_2, E(k_3, p)))$, it would have been impossible to use 3DES implementations for systems that required compatibility with DES. This is particularly important for hardware implementations, where it is not always possible to provide a secondary, regular "single DES" interface next to the primary 3DES interface.

Some attacks on 3DES are known, reducing their effective security. While breaking 3DES with the first keying option is currently impractical, 3DES is a poor choice for any modern cryptosystem. The security margin is already small, and continues to shrink as cryptographic attacks improve and processing power grows.

Far better alternatives, such as AES, are available. Not only are they more secure than 3DES, they are also generally much, much faster. On the same hardware and in the same *mode of operation* (we'll explain what that means in the next chapter), AES-128 only takes 12.6 cycles per byte, while 3DES takes up to 134.5 cycles per byte. [Dai] Despite being worse from a security point of view, it is literally an order of magnitude slower.

While more iterations of DES might increase the security margin, they aren't used in practice. First of all, the process has never been standardized beyond three iterations. Also, the performance only becomes worse as you add more iterations. Finally, increasing the key bits has diminishing security returns, only increasing the security level of the resulting algorithm by a smaller amount as the number of key bits increases. While 3DES with keying option 1 has a key length of 168 bits, the effective security level is estimated at only 112 bits.

Even though 3DES is significantly worse in terms of performance and slightly worse in terms of security, 3DES is still the workhorse of the financial industry. With a plethora of standards already in existence and new ones continuing to be created, in such an extremely technologically conservative industry where Fortran and Cobol still reign supreme on massive mainframes, it will probably continue to be used for many years to come, unless there are some large cryptanalytic breakthroughs that threaten the security of 3DES.

## 6.4 Remaining problems

Even with block ciphers, there are still some unsolved problems.

For example, we can only send messages of a very limited length: the block length of the block cipher. Obviously, we'd like to be able to send much larger messages, or, ideally, streams of indeterminate size. We'll address this problem with a *stream cipher* (page 41).

Although we have reduced the key size drastically (from the total size of all data ever sent under a one-time pad scheme versus a few bytes for most block ciphers), we still need to address the issue of agreeing on those few key bytes, potentially over an insecure channel. We'll address this problem in a later chapter with a *key exchange protocol* (page 81).

# 7

# Stream ciphers

## 7.1 Description

A *stream cipher* is a *symmetric-key encryption* algorithm that encrypts a stream of bits. Ideally, that stream could be as long as we'd like; real-world *stream cipher*s have limits, but they are normally sufficiently large that they don't pose a practical problem.

## 7.2 A naive attempt with block ciphers

Let's try to build a *stream cipher* using the tools we already have. Since we already have block ciphers, we could simply divide an incoming stream into different blocks, and encrypt each block:

$$\begin{array}{cccc} \underbrace{\texttt{abcdefgh}} & \underbrace{\texttt{ijklmno}} & \underbrace{\texttt{pqrstuvw}} & \ldots \\ \downarrow & \downarrow & \downarrow & \\ \overbrace{\texttt{APOHGMMW}} & \overbrace{\texttt{PVMEHQOM}} & \overbrace{\texttt{MEEZSNFM}} & \ldots \end{array}$$

This scheme is called *ECB mode* (Electronic Code Book Mode), and it is one of the many ways that block ciphers can be used to construct stream ciphers. Unfortunately, while being very common in home-grown cryptosystems, it poses very serious security flaws. For example, in ECB mode, identical input blocks will always map to identical output blocks:

$$\begin{array}{cccc} \underbrace{\texttt{abcdefgh}} & \underbrace{\texttt{abcdefgh}} & \underbrace{\texttt{abcdefgh}} & \ldots \\ \downarrow & \downarrow & \downarrow & \\ \overbrace{\texttt{APOHGMMW}} & \overbrace{\texttt{APOHGMMW}} & \overbrace{\texttt{APOHGMMW}} & \ldots \end{array}$$

At first, this might not seem like a particularly serious problem. Assuming the block cipher is secure, it doesn't look like an attacker would be able to decrypt anything. By dividing the ciphertext stream up into blocks, an attacker would only be able to see that a ciphertext block, and therefore a plaintext block, was repeated.

We'll now illustrate the many flaws of *ECB mode* with two attacks. First, we'll exploit the fact that repeating plaintext blocks result in repeating ciphertext blocks, by visually inspecting an encrypted image. Then, we'll demonstrate that attackers can often decrypt messages encrypted in *ECB mode* by communicating with the person performing the encryption.

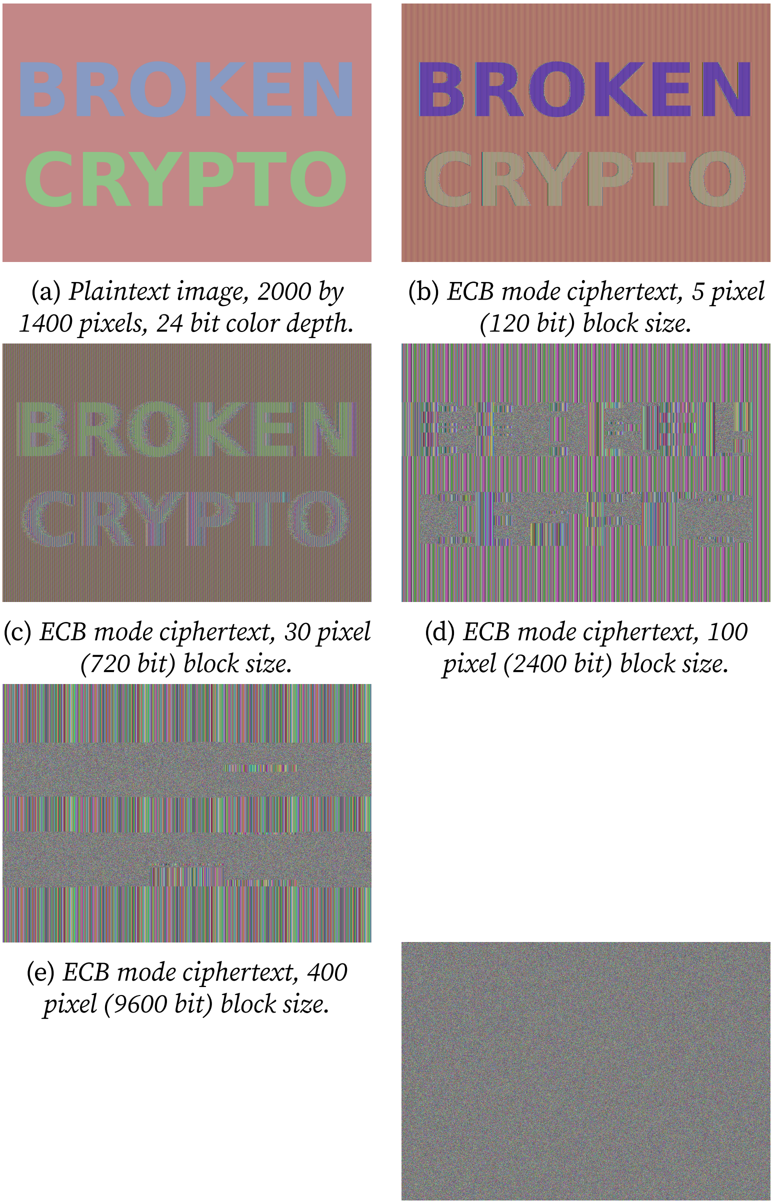
## Visual inspection of an encrypted stream

To demonstrate that this is, in fact, a serious problem, we'll use a simulated block cipher of various block sizes and apply it to an image[1]. We'll then visually inspect the different outputs.

Because identical blocks of pixels in the plaintext will map to identical blocks of pixels in the ciphertext, the global structure of the image is largely preserved.

As you can see, the situation appears to get slightly better with larger block sizes, but the fundamental problem still remains: the macrostructure of the image remains visible in

[1] This particular demonstration only works on uncompressed bitmaps. For other media, the effect isn't significantly less damning: it's just less visual.



(a) *Plaintext image, 2000 by 1400 pixels, 24 bit color depth.*

(b) *ECB mode ciphertext, 5 pixel (120 bit) block size.*

(c) *ECB mode ciphertext, 30 pixel (720 bit) block size.*

(d) *ECB mode ciphertext, 100 pixel (2400 bit) block size.*

(e) *ECB mode ciphertext, 400 pixel (9600 bit) block size.*

(f) *Ciphertext under idealized encryption.*

Figure 7.1: Plaintext image with ciphertext images under idealized encryption and *ECB mode* encryption with various block sizes. Information about the macro-structure of the image clearly leaks. This becomes less apparent as block sizes increase, but only at block sizes far larger than typical block ciphers. Only the first block size (Figure 7.1f, a block size of 5 pixels or 120 bits) is realistic.

all but the most extreme block sizes. Furthermore, all but the smallest of these block sizes are unrealistically large. For an uncompressed bitmap with three color channels of 8 bit depth, each pixel takes 24 bits to store. Since the block size of AES is only 128 bits, that would equate to $\frac{128}{24}$ or just over 5 pixels per block. That's significantly fewer pixels per block than the larger block sizes in the example. But AES is the workhorse of modern block ciphers—it can't be at fault, certainly not because of an insufficient block size.

When we look at a picture of what would happen with an idealized encryption scheme, we notice that it looks like random noise. Keep in mind that "looking like random noise" doesn't mean something is properly encrypted: it just means that we can't inspect it using methods this trivial.

## Encryption oracle attack

In the previous section, we've focused on how an attacker can inspect a ciphertext encrypted using *ECB mode*. That's a *passive*, ciphertext-only attack. It's passive because the attacker doesn't really interfere in any communication; they're simply examining a ciphertext. In this section, we'll study a different, *active* attack, where the attacker actively communicates with their target. We'll see how the active attack can enable an attacker to decrypt ciphertexts encrypted using *ECB mode*.

To do this, we'll introduce a new concept called an *oracle*. Formally defined *oracle*s are used in the study of computer science, but for our purposes it's sufficient to just say that an *oracle* is something that will compute some particular function for you.

In our case, the *oracle* will perform a specific encryption for the attacker, which is why it's called an *encryption oracle*. Given some data $A$ chosen by the attacker, the *oracle* will encrypt that data, followed by a secret suffix $S$, in *ECB mode*. Or, in symbols:

$$C = ECB(E_k, A \| S)$$

The secret suffix $S$ is specific to this system. The attacker's goal is to decrypt it. We'll see that being able to encrypt other messages surprisingly allows the attacker to decrypt the suffix. This *oracle* might seem artificial, but is quite common in practice. A simple example would be a cookie encrypted with ECB, where the prefix $A$ is a name or an e-mail address field, controlled by the attacker.

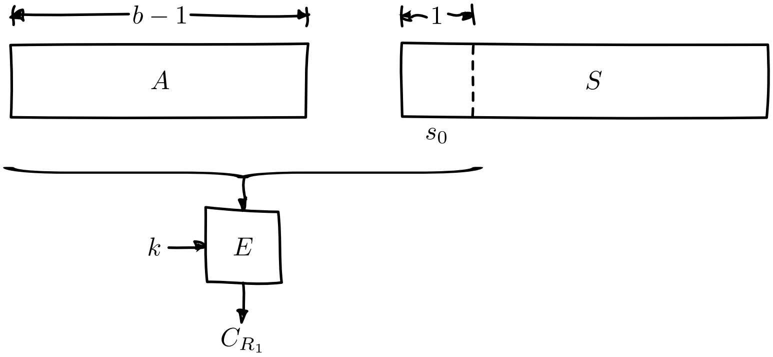
You can see why the concept of an *oracle* is important here: the attacker would not be able to compute $C$ themselves, since they do not have access to the encryption key $k$ or the secret suffix $S$. The goal of the *oracle* is for those values to remain secret, but we'll see how an attacker will be able to recover the secret suffix $S$ (but not the key $k$) anyway. The attacker does this by inspecting the ciphertext $C$ for many carefully chosen values of the attacker-chosen prefix $A$.

Assuming that an attacker would have access to such an *oracle* might seem like a very artificial scenario. It turns out that in practice, a lot of software can be tricked into behaving like one. Even if an attacker can't control the real software as precisely as they can query an *oracle*, the attacker generally isn't thwarted. Time is on their side: they only have to convince the software to give the answer they want *once*. Systems where part of the message is secret and part of the message can be influenced by the attacker are actually very common, and, unfortunately, so is *ECB mode*.

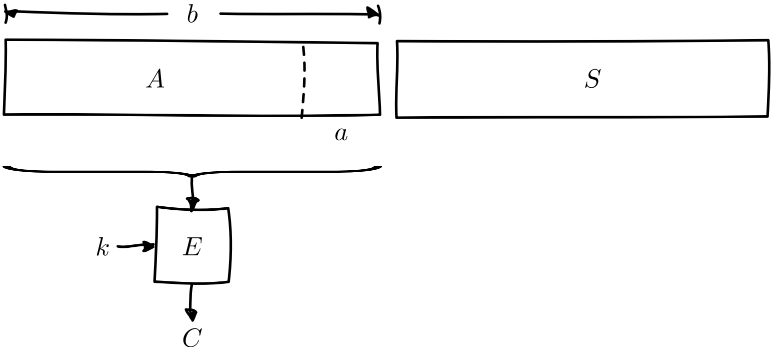
## Decrypting a block using the oracle

The attacker starts by sending in a plaintext $A$ that's just one byte shorter than the block size. That means the block that's being encrypted will consist of those bytes, plus the first byte of $S$, which we'll call $s_0$. The attacker remembers the encrypted block. They don't know the value of $s_0$ yet, but now they do know the value of the first encrypted block: $E_k(A \| s_0)$. In the illustration, this is block $C_{R1}$:

Then, the attacker tries a full-size block, trying all pos-



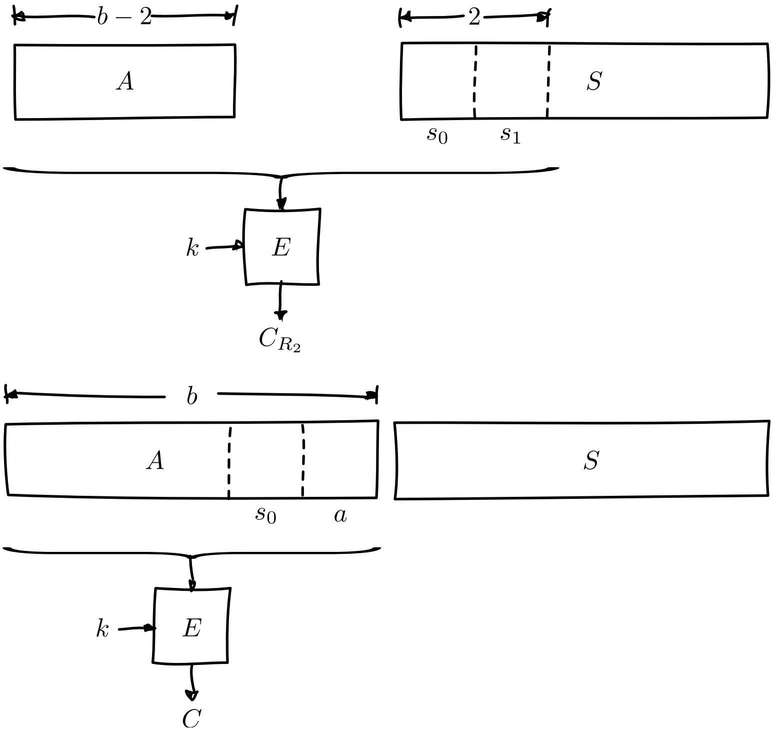
sible values for the final byte. Eventually, they'll find the value of $s_0$; they know the guess is correct because the resulting ciphertext block will match the ciphertext block $C_{R1}$ they remembered earlier.



The attacker can repeat this for the penultimate byte. They submit a plaintext $A$ that's two bytes shorter than the block size. The *oracle* will encrypt a first block consisting of that $A$ followed by the first two bytes of the secret suffix, $s_0s_1$. The attacker remembers that block.

Since the attacker already knows $s_0$, they try $A \| s_0$ followed by all possible values of $s_1$. Eventually they'll guess correctly, which, again, they'll know because the ciphertext blocks match:

The attacker can then rinse and repeat, eventually decrypting an entire block. This allows them to brute-force a



block in $p \cdot b$ attempts, where $p$ is the number of possible values for each byte (so, for 8-bit bytes, that's $2^8 = 256$) and $b$ is the block size. This is much better than a regular brute-force attack, where an attacker has to try all of the possible blocks, which would be:

$$\underbrace{p \cdot p \ldots \cdot p}_{b \text{ positions}} = p^b$$

For a typical block size of 16 bytes (or 128 bits), brute forcing would mean trying $256^{16}$ combinations. That's a huge, 39-digit number. It's so large that trying all of those combinations is considered impossible. An ECB *encryption oracle* allows an attacker to do it in at most $256 \cdot 16 = 4096$ tries, a far more manageable number.

### Conclusion

In the real world, block ciphers are used in systems that encrypt large amounts of data all the time. We've seen that when using *ECB mode*, an attacker can both analyze ciphertexts to recognize repeating patterns, and even decrypt messages when given access to an *encryption oracle*.

Even when we use idealized block ciphers with unrealistic properties, such as block sizes of more than a thousand bits, an attacker ends up being able to decrypt the ciphertexts. Real world block ciphers only have more limitations than our idealized examples, such as much smaller block sizes.

We aren't even taking into account any potential weaknesses in the block cipher. It's not AES (or our test block ciphers) that cause this problem, it's our ECB construction. Clearly, we need something better.

## 7.3 Block cipher modes of operation

One of the more common ways of producing a *stream cipher* is to use a block cipher in a particular configuration. The compound system behaves like a *stream cipher*. These configurations are commonly called *mode of operation*s. They aren't specific to a particular block cipher.

*ECB mode*, which we've just seen, is the simplest such *mode of operation*. The letters ECB stand for electronic code book[2]. For reasons we've already gone into, *ECB mode* is very ineffective. Fortunately, there are plenty of other choices.

## 7.4 CBC mode

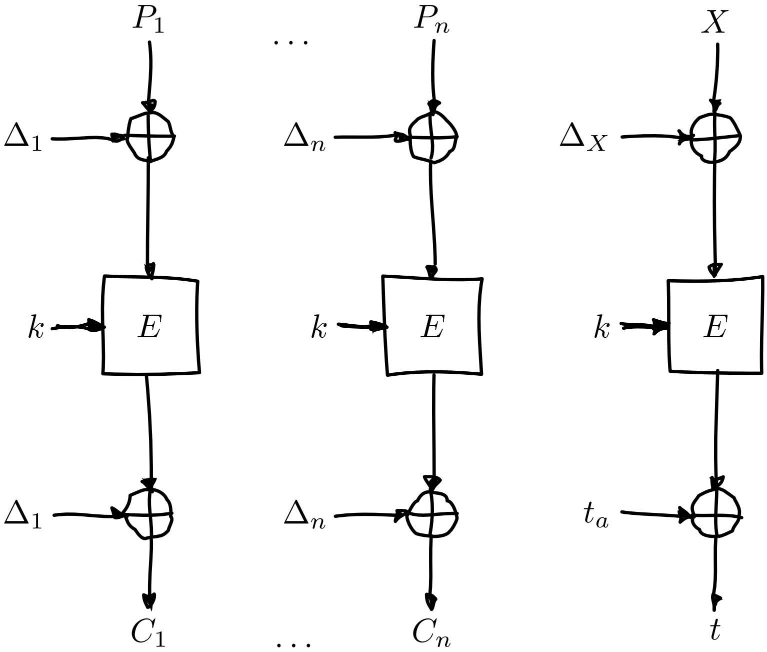
*CBC mode*, which stands for cipher block chaining, is a very common *mode of operation* where plaintext blocks are XORed

[2] Traditionally, modes of operation seem to be referred to by a three-letter acronym.

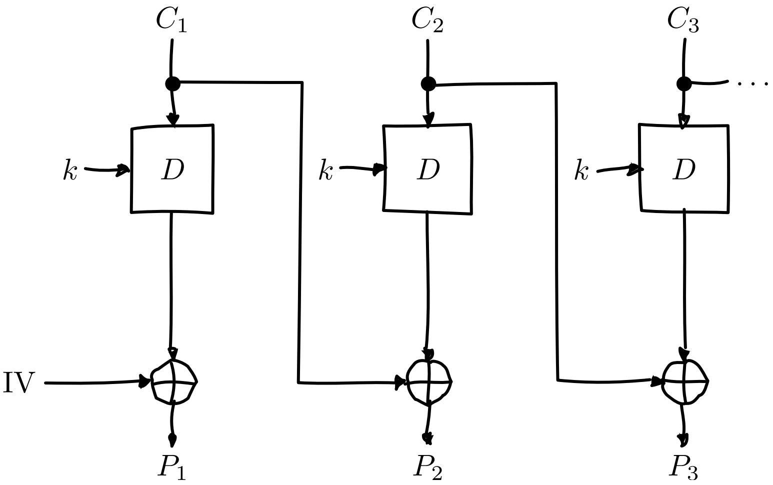
with the previous ciphertext block before being encrypted by the block cipher.

Of course, this leaves us with a problem for the first plaintext block: there is no previous ciphertext block to XOR it with. Instead, we pick an IV: a random number that takes the place of the "first" ciphertext in this construction. *initialization vector*s also appear in many other algorithms. An *initialization vector* should be unpredictable; ideally, they will be cryptographically random. They do not have to be secret: IVs are typically just added to ciphertext messages in plaintext. It may sound contradictory that something has to be unpredictable, but doesn't have to be secret; it's important to remember that an attacker must not be able to predict *ahead of time* what a given IV will be. We will illustrate this later with an attack on predictable CBC IVs.

The following diagram demonstrates encryption in *CBC mode*:



Decryption is the inverse construction, with block ciphers in decryption mode instead of encryption mode:



While *CBC mode* itself is not inherently insecure (unlike *ECB mode*), its particular use in TLS 1.0 was. This eventually led to the BEAST attack, which we'll cover in more detail in the section on SSL/TLS. The short version is that instead of using unpredictable *initialization vectors*, for example by choosing random IVs, the standard used the previous ciphertext block as the IV for the next message. Unfortunately, it turns out that attackers figured out how to exploit that property.

## 7.5 Attacks on CBC mode with predictable IVs

Suppose there's a database that stores secret user information, like medical, payroll or even criminal records. In order to protect that information, the server that handles it encrypts it using a strong block cipher in *CBC mode* with a fixed key. For now, we'll assume that that server is secure, and there's no way to get it to leak the key.

Mallory gets a hold of all of the rows in the database. Perhaps she did it through a SQL injection attack, or maybe with a little social engineering.[3] Everything is supposed to remain secure: Mallory only has the ciphertexts, but she doesn't have the secret key.

Mallory wants to figure out what Alice's record says. For simplicity's sake, let's say there's only one ciphertext block. That means Alice's ciphertext consists of an IV and one ciphertext block.

Mallory can still try to use the application as a normal user, meaning that the application will encrypt some data of Mallory's choosing and write it to the database. Suppose that through a bug in the server, Mallory can predict the IV that will be used for her ciphertext. Perhaps the server always uses the same IV for the same person, or always uses an all-zero IV, or...

Mallory can construct her plaintext using Alice's IV $IV_A$ (which Mallory can see) and her own predicted IV $IV_M$. She makes a guess $G$ as to what Alice's data could be. She asks the server to encrypt:

$$P_M = IV_M \oplus IV_A \oplus G$$

The server dutifully encrypts that message using the predicted IV $IV_M$. It computes:

$$\begin{aligned} C_M &= E(k, IV_M \oplus P_M) \\ &= E(k, IV_M \oplus (IV_M \oplus IV_A \oplus G)) \\ &= E(k, IV_A \oplus G) \end{aligned}$$

That ciphertext, $\mathrm{C_M}$, is exactly the ciphertext block Alice would have had if her plaintext block was G. So, depending on what the data is, Mallory has figured out if Alice has a

[3] Social engineering means tricking people into things they shouldn't be doing, like giving out secret keys, or performing certain operations. It's usually the most effective way to break otherwise secure cryptosystems.

criminal record or not, or perhaps some kind of embarrassing disease, or some other issue that Alice really expected the server to keep secret.

Lessons learned: don't let IVs be predictable. Also, don't roll your own cryptosystems. In a secure system, Alice and Mallory's records probably wouldn't be encrypted using the same key.

## 7.6 Attacks on CBC mode with the key as the IV

Many CBC systems set the key as the *initialization vector*. This seems like a good idea: you always need a shared secret key already anyway. It yields a nice performance benefit, because the sender and the receiver don't have to communicate the IV explicitly, they already know the key (and therefore the IV) ahead of time. Plus, the key is definitely unpredictable because it's secret: if it were predictable, the attacker could just predict the key directly and already have won. Conveniently, many block ciphers have block sizes that are the same length or less than the key size, so the key is big enough.

This setup is completely insecure. If Alice sends a message to Bob, Mallory, an active adversary who can intercept and modify the message, can perform a chosen ciphertext attack to recover the key.

Alice turns her plaintext message $P$ into three blocks $P_1 P_2 P_3$ and encrypts it in *CBC mode* with the secret key $k$ and also uses $k$ as the IV. She gets a three block ciphertext $C = C_1 C_2 C_3$, which she sends to Bob.

Before the message reaches Bob, Mallory intercepts it. She modifies the message to be $C' = C_1 Z C_1$, where $Z$ is a block filled with null bytes (value zero).

Bob decrypts $C'$, and gets the three plaintext blocks

$P'_1, P'_2, P'_3$:

$$\begin{aligned}
P'_1 &= D(k, C_1) \oplus IV \\
&= D(k, C_1) \oplus k \\
&= P_1 \\
P'_2 &= D(k, Z) \oplus C_1 \\
&= R \\
P'_3 &= D(k, C_1) \oplus Z \\
&= D(k, C_1) \\
&= P_1 \oplus IV
\end{aligned}$$

$R$ is some random block. Its value doesn't matter.

Under the chosen-ciphertext attack assumption, Mallory recovers that decryption. She is only interested in the first block ($P'_1 = P_1$) and the third block ($P'_3 = P_1 \oplus IV$). By XORing those two together, she finds $(P_1 \oplus IV) \oplus P_1 = IV$. But, the IV is the key, so Mallory successfully recovered the key by modifying a single message.

Lesson learned: don't use the key as an IV. Part of the fallacy in the introduction is that it assumed secret data could be used for the IV, because it only had to be unpredictable. That's not true: "secret" is just a different requirement from "not secret", not necessarily a *stronger* one. It is not generally okay to use secret information where it isn't required, precisely because if it's not supposed to be secret, the algorithm may very well treat it as non-secret, as is the case here. There *are* plenty of systems where it is okay to use a secret where it isn't required. In some cases you might even get a stronger system as a result, but the point is that it is not generally true, and depends on what you're doing.

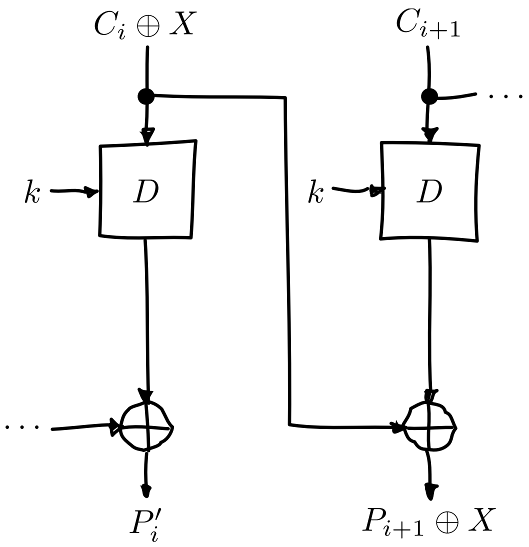
## 7.7 CBC bit flipping attacks

An interesting attack on *CBC mode* is called a bit flipping attack. Using a CBC bit flipping attack, attackers can modify ciphertexts encrypted in *CBC mode* so that it will have a predictable effect on the plaintext.

This may seem like a very strange definition of "attack" at first. The attacker will not even attempt to decrypt any messages, but they will just be flipping some bits in a plaintext. We will demonstrate that the attacker can turn the ability to flip some bits in the plaintext into the ability to have the plaintext say *whatever they want it to say*, and, of course, that can lead to very serious problems in real systems.

Suppose we have a CBC encrypted ciphertext. This could be, for example, a cookie. We take a particular ciphertext block, and we flip some bits in it. What happens to the plaintext?

When we "flip some bits", we do that by XORing with a sequence of bits, which we'll call $X$. If the corresponding bit in $X$ is 1, the bit will be flipped; otherwise, the bit will remain the same.



When we try to decrypt the ciphertext block with the flipped bits, we will get indecipherable[4] nonsense. Remem-

[4] Excuse the pun.

ber how CBC decryption works: the output of the block cipher is XORed with the previous ciphertext block to produce the plaintext block. Now that the input ciphertext block $C_i$ has been modified, the output of the block cipher will be some random unrelated block, and, statistically speaking, nonsense. After being XORed with that previous ciphertext block, it will still be nonsense. As a result, the produced plaintext block is still just nonsense. In the illustration, this unintelligible plaintext block is $P_i'$.

However, in the block *after* that, the bits we flipped in the ciphertext will be flipped in the plaintext as well! This is because, in CBC decryption, ciphertext blocks are decrypted by the block cipher, and the result is XORed with the previous ciphertext block. But since we modified the previous ciphertext block by XORing it with $X$, the plaintext block $P_{i+1}$ will also be XORed with $X$. As a result, the attacker completely controls that plaintext block $P_{i+1}$, since they can just flip the bits that aren't the value they want them to be.

TODO: add previous illustration, but mark the path X takes to influence P prime {i + 1} in red or something

This may not sound like a huge deal at first. If you don't know the plaintext bytes of that next block, you have no idea which bits to flip in order to get the plaintext you want.

To illustrate how attackers can turn this into a practical attack, let's consider a website using cookies. When you register, your chosen user name is put into a cookie. The website encrypts the cookie and sends it to your browser. The next time your browser visits the website, it will provide the encrypted cookie; the website decrypts it and knows who you are.

An attacker can often control at least part of the plaintext being encrypted. In this example, the user name is part of the plaintext of the cookie. Of course, the website just lets you provide whatever value for the user name you want at registration, so the attacker can just add a very long string of `Z` bytes to their user name. The server will happily encrypt such a cookie, giving the attacker an encrypted ciphertext that matches a plaintext with many such `Z` bytes in them. The plaintext getting modified will then probably be part of that sequence of `Z` bytes.

An attacker may have some target bytes that they'd like to see in the decrypted plaintext, for example, `;admin=1;`. In order to figure out which bytes they should flip (so, the value of $X$ in the illustration), they just XOR the filler bytes (~ZZZ~...) with that target. Because two XOR operations with the same value cancel each other out, the two filler values (~ZZZ~...) will cancel out, and the attacker can expect to see `;admin=1;` pop up in the next plaintext block:

$$\begin{aligned} P'_{i+1} &= P_{i+1} \oplus X \\ &= P_{i+1} \oplus \texttt{ZZZZZZZZZ} \oplus ;\texttt{admin} = \texttt{1}; \\ &= \texttt{ZZZZZZZZZ} \oplus \texttt{ZZZZZZZZZ} \oplus ;\texttt{admin} = \texttt{1}; \\ &= ;\texttt{admin} = \texttt{1}; \end{aligned}$$

This attack is another demonstration of an important cryptographic principle: encryption is not authentication! It's virtually never sufficient to simply encrypt a message. It *may* prevent an attacker from reading it, but that's often not even necessary for the attacker to be able to modify it to say whatever they want it to. This particular problem would be solved by also securely authenticating the message. We'll see how you can do that later in the book; for now, just remember that we're going to need authentication in order to produce secure cryptosystems.

## 7.8 Padding

So far, we've conveniently assumed that all messages just happened to fit exactly in our system of block ciphers, be it CBC or ECB. That means that all messages happen to be a multiple of the block size, which, in a typical block cipher such as AES, is 16 bytes. Of course, real messages can be of arbitrary length. We need some scheme to make them fit. That process is called padding.

### Padding with zeroes (or some other pad byte)

One way to pad would be to simply append a particular byte value until the plaintext is of the appropriate length. To undo the padding, you just remove those bytes. This scheme has an obvious flaw: you can't send messages that end in that particular byte value, or you will be unable to distinguish between padding and the actual message.

### PKCS#5/PKCS#7 padding

A better, and much more popular scheme, is PKCS#5/PKCS#7 padding.

PKCS#5, PKCS#7 and later CMS padding are all more or less the same idea[5]. Take the number of bytes you have to pad, and pad them with that many times the byte with that value. For example, if the block size is 8 bytes, and the last block has the three bytes `12 34 45`, the block becomes `12 34 45 05 05 05 05 05` after padding.

If the plaintext happened to be exactly a multiple of the block size, an entire block of padding is used. Otherwise, the recipient would look at the last byte of the plaintext, treat it as a padding length, and almost certainly conclude the message was improperly padded.

This scheme is described in [Hou].

## 7.9 CBC padding attacks

We can refine CBC bit flipping attacks to trick a recipient into decrypting arbitrary messages!

As we've just discussed, *CBC mode* requires padding the message to a multiple of the block size. If the padding is incorrect, the recipient typically rejects the message, saying

[5] Technically, PKCS#5 padding is only defined for 8 byte block sizes, but the idea clearly generalizes easily, and it's also the most commonly used term.

that the padding was invalid. We can use that tiny bit of information about the padding of the plaintext to iteratively decrypt the entire message.

The attacker will do this, one ciphertext block at a time, by trying to get an entire plaintext block worth of valid padding. We'll see that this tells them the decryption of their target ciphertext block, under the block cipher. We'll also see that you can do this efficiently and iteratively, just from that little leak of information about the padding being valid or not.

It may be helpful to keep in mind that a CBC padding attack does not actually attack the padding for a given message; instead the attacker will be *constructing* paddings to decrypt a message.

To mount this attack, an attacker only needs two things:

1. A target ciphertext to decrypt

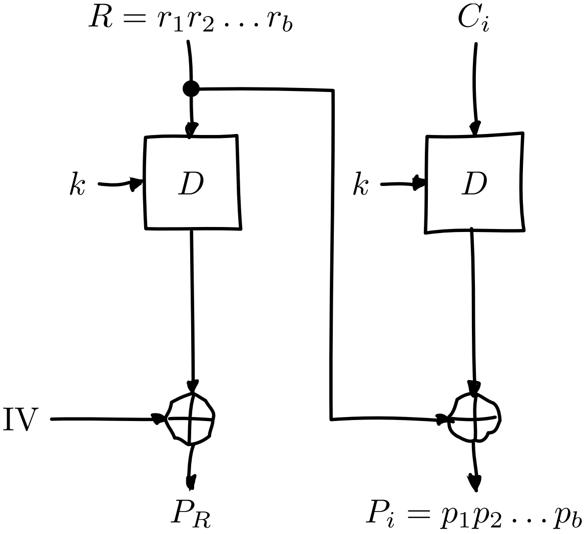
2. A *padding oracle*: a function that takes ciphertexts and tells the attacker if the padding was correct

As with the ECB *encryption oracle*, the availability of a padding oracle may sound like a very unrealistic assumption. The massive impact of this attack proves otherwise. For a long time, most systems did not even attempt to hide if the padding was valid or not. This attack remained dangerous for a long time after it was originally discovered, because it turns out that in many systems it is extremely difficult to actually hide if padding is valid or not. We will go into this problem in more detail both in this chapter and in later chapters.

In this chapter, we'll assume that PKCS#5/PKCS#7 padding is being used, since that's the most popular option. The attack is general enough to work on other kinds of padding, with minor modifications.

## Decrypting the first byte

The attacker fills a block with arbitrary bytes $R = r_1, r_2 \ldots r_b$. They also pick a target block $C_i$ from the ciphertext that they'd like to decrypt. The attacker asks the padding oracle if the plaintext of $R \| C_i$ has valid padding. Statistically speaking, such a random plaintext probably won't have valid padding: the odds are in the half-a-percent ballpark. If by pure chance the message happens to already have valid padding, the attacker can simply skip the next step.



Next, the attacker tries to modify the message so that it does have valid padding. They can do that by indirectly modifying the last byte of the plaintext: eventually that byte will be `01`, which is always valid padding. In order to modify the last byte of a plaintext block, the attacker modifies the last byte of the *previous* ciphertext block. This works exactly like it did with CBC bit flipping attacks. That previous ciphertext block is the block $R$, so the byte being modified is the last byte of $R$, $r_b$.

The attacker tries all possible values for that last byte. There are several ways of doing that: modular addition, XORing it with all values up to 256, or even picking randomly; the only thing that matters is that the attacker tries all of them. Eventually, the padding oracle will report that for some ciphertext block $R$, the decrypted plaintext of $R||C_i$ has valid padding.

## Discovering the padding length

The oracle has just told the attacker that for our chosen value of $R$, the plaintext of $R||C_i$ has valid padding. Since we're working with PKCS#5 padding, that means that the plaintext block $P_i$ ends in one of the following byte sequences:

- `01`
- `02 02`
- `03 03 03`
- …

The first option (`01`) is much more likely than the others, since it only requires one byte to have a particular value. The attacker is modifying that byte to take *every* possible value, so it is quite likely that they happened to stumble upon `01`. All of the other valid padding options not only require that byte to have some particular value, but also one or more other bytes. For an attacker to be guaranteed a message with a valid `01` padding, they just have to try every possible byte. For an attacker to end up with a message with a valid `02 02` padding, they have to try every possible byte *and* happen to have picked a combination of $C$ and $R$ that causes the plaintext to have a `02` in that second-to-last position. (To rephrase: the second-to-last byte of the decryption of the ciphertext block, XORed with the second-to-last byte of $R$, is `02`.)

In order to successfully decrypt the message, we still need to figure out which one of those options is the actual value of the padding. To do that, we try to discover the length of the padding by modifying bytes starting at the left-hand side of $P_i$ until the padding becomes invalid again. As with everything else in this attack, we modify those bytes in $P_i$ by modifying the equivalent bytes in our chosen block $R$. As soon as padding breaks, you know that the last byte you modified was part of the valid padding, which tells you how many padding bytes there are. Since we're using PKCS#5 padding, that also tells you what their value is.

Let's illustrate this with an example. Suppose we've successfully found some block $R$ so that the plaintext of $R||C_i$ has valid padding. Let's say that padding is `03 03 03`. Normally, the attacker wouldn't know this; the point of this procedure is to discover what that padding is. Suppose the block size is 8 bytes. So, we (but not the attacker) know that $P_i$ is currently:

$$p_0 p_1 p_2 p_3 p_4 \texttt{030303}$$

In that equation, $p_0 \ldots$ are some bytes of the plaintext. Their actual value doesn't matter: the only thing that matters is that they're not part of the padding. When we modify the first byte of $R$, we'll cause a change in the first byte of $P_i$, so that $p_0$ becomes some other byte $p'_0$:

$$p'_0 p_1 p_2 p_3 p_4 \texttt{030303}$$

As you can see, this doesn't affect the validity of the padding. It also does not affect $p_1$, $p_2$, $p_3$ or $p_4$. However, when we continue modifying subsequent bytes, we will eventually hit a byte that *is* part of the padding. For example, let's say we turn that first `03` into `02` by modifying $R$. $P_i$ now looks like this:

$$p'_0 p'_1 p'_2 p'_3 p'_4 \texttt{020303}$$

Since `02 03 03` isn't valid PKCS#5 padding, the server will reject the message. At that point, we know that once we modify six bytes, the padding breaks. That means the sixth byte is the first byte of the padding. Since the block is 8 bytes long, we know that the padding consists of the sixth, seventh and eighth bytes. So, the padding is three bytes long, and, in PKCS#5, equal to `03 03 03`.

A clever attacker who's trying to minimize the number of oracle queries can leverage the fact that longer valid padding becomes progressively more rare. They can do this by starting from the penultimate byte instead of the beginning of the block. The advantage to this method is that short paddings (which are more common) are detected more quickly. For example, if the padding is `0x01` and an attacker starts modifying the penultimate byte, they only need one query to learn what the padding was. If the penultimate byte is changed to any other value and the padding is still valid, the padding must be `0x01`. If the padding is not valid, the padding must be at least `0x02 0x02`. So, they go back to the original block and start modifying the third byte from the back. If that passes, the padding was indeed `0x02 0x02`, otherwise the padding must be at least `0x03 0x03 0x03`. The process repeats until they've found the correct length. This is a little trickier to implement; you can't just keep modifying the same block (if it's mutable), and you're waiting for the oracle to fail instead of pass, which can be confusing. But other than being faster at the cost of being slightly more complex, this technique is equivalent to the one described above.

For the next section, we'll assume that it was just `01`, since that is the most common case. The attack doesn't really change depending on the length of the padding. If you guess more bytes of padding correctly, that just means that there are fewer remaining bytes you will have to guess manually. (This will become clear once you understand the rest of the attack.)

## Decrypting one byte

At this point, the attacker has already successfully decrypted the last byte of the target block of ciphertext! Actually, we've decrypted as many bytes as we have valid padding; we're just assuming the worst case scenario where there is only a single byte. How? The attacker knows that the last byte of the decrypted ciphertext block $C_i$ (we'll call that byte $D(C_i)[b]$), XORed with the iteratively found value $r_b$, is `01`:

$$D(C_i)[b] \oplus r_b = \texttt{01}$$

By moving the XOR operation to the other side, the attacker gets:

$$D(C_i)[b] = \texttt{01} \oplus r_b$$

The attacker has now tricked the receiver into revealing the value of the last byte of the block cipher decryption of $C_i$.

## Decrypting subsequent bytes

Next, the attacker tricks the receiver into decrypting the next byte. Remember the previous equation, where we reasoned that the last byte of the plaintext was `01`:

$$D(C_i)[b] \oplus r_b = \texttt{01}$$

Now, we'd like to get that byte to say `02`, to produce an *almost* valid padding: the last byte would be correct for a 2-byte PKCS#5 padding (`02 02`), but that second-to-last byte probably isn't `02` yet. To do that, we XOR with `01` to cancel the `01` that's already there (since two XORs with the same value cancel each other out), and then we XOR with `02` to get `02`:

$$\begin{aligned} D(C_i)[b] \oplus r_b \oplus \texttt{01} \oplus \texttt{02} &= \texttt{01} \oplus \texttt{01} \oplus \texttt{02} \\ &= \texttt{02} \end{aligned}$$

So, to produce a value of `02` in the final position of the decrypted plaintext, the attacker replaces $r_b$ with:

$$r'_b = r_b \oplus \texttt{01} \oplus \texttt{02}$$

This accomplishes the goal of almost valid padding. Then, they try all possible values for the second-to-last byte (index $b - 1$). Eventually, one of them will cause the message to have valid padding. Since we modified the random block so that the final byte of the plaintext will be `02`, the only byte in the second-to-last position that can cause valid padding is `02` as well. Using the same math as above, the attacker has recovered the second-to-last byte.

Then, it's just rinse and repeat. The last two bytes are modified to create an almost-valid padding of `03 03`, then the third byte from the right is modified until the padding is valid, and so on. Repeating this for all the bytes in the block means the attacker can decrypt the entire block; repeating it for different blocks means the attacker can read the entire message.

This attack has proven to be very subtle and hard to fix. First of all, messages should be authenticated, as well as encrypted. That would cause modified messages to be rejected. However, many systems decrypt (and remove padding) before authenticating the message; so the information about the padding being valid or not has already leaked. We will discuss secure ways of authenticating messages later in the book.

You might consider just getting rid of the "invalid padding" message; declaring the message invalid without specifying *why* it was invalid. That turns out to only be a partial solution for systems that decrypt before authenticating. Those systems would typically reject messages with an invalid padding *slightly faster* than messages with a valid padding. After all, they didn't have to do the authentication step: if the padding is invalid, the message can't possibly be valid. An attack that leaks secret information through timing differences is called a *timing attack*, which is a special case of a *side-channel attack*: attacks on the practical implementation of a cryptosystem rather than its "perfect" abstract representation. We will talk about these kinds of attacks more later in the book.

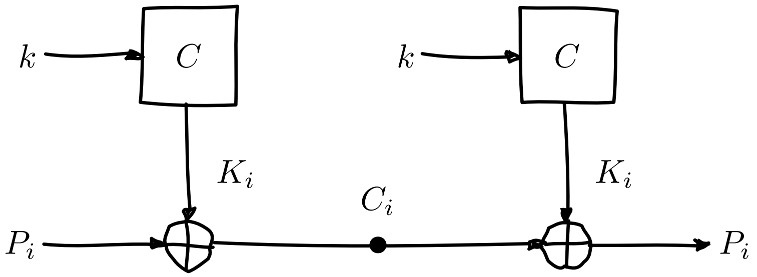
That discrepancy was commonly exploited as well. By measuring how long it takes the recipient to reject the message, the attacker can tell if the recipient performed the authentication step. That tells them if the padding was correct or not, providing the padding oracle to complete the attack.

The principal lesson learned here is, again, not to design your own cryptosystems. The main way to avoid this particular problem is by performing constant time authentication, and authenticating the ciphertext before decrypting it. We will talk more about this in a later chapter on message authentication.

## 7.10 Native stream ciphers

In addition to block ciphers being used in a particular *mode of operation*, there are also "native" *stream cipher*s algorithms that are designed from the ground up to be a *stream cipher*.

The most common type of *stream cipher* is called a *synchronous* stream cipher. These algorithms produce a long stream of pseudorandom bits from a secret symmetric key. This stream, called the keystream, is then XORed with the plaintext to produce the ciphertext. Decryption is the identical operation as encryption, just repeated: the keystream is produced from the key, and is XORed with the ciphertext to produce the plaintext.



You can see how this construction looks quite similar to a one-time pad, except that the truly random one-time pad has been replaced by a pseudorandom *stream cipher*.

There are also *asynchronous* or *self-synchronizing* *stream cipher*s, where the previously produced ciphertext bits are used to produce the current keystream bit. This has the interesting consequence that a receiver can eventually recover if some ciphertext bits are dropped. This is generally not considered to be a desirable property anymore in modern cryptosystems, which instead prefer to send complete, authenticated messages. As a result, these *stream cipher*s are very rare, and we don't talk about them explicitly in this book. Whenever someone says "stream cipher", it's safe to assume they mean the synchronous kind.

Historically, native *stream cipher*s have had their issues. NESSIE, an international competition for new cryptographic primitives, for example, did not result in any new *stream cipher*s, because all of the participants were broken before the competition ended. RC4, one of the most popular native *stream cipher*s, has had serious known issues for years. By comparison, some of the constructions using block ciphers seem bulletproof.

Fortunately, more recently, several new cipher algorithms provide new hope that we can get practical, secure and performant *stream cipher*s.

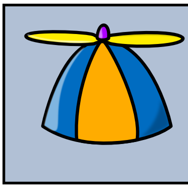
## 7.11 RC4

By far the most common native *stream cipher* in common use on desktop and mobile devices is RC4.

RC4 is sometimes also called ARCFOUR or ARC4, which stands for *alleged* RC4. While its source code has been leaked and its implementation is now well-known, RSA Security (the company that authored RC4 and still holds the RC4 trademark) has never acknowledged that it is the real algorithm.

It quickly became popular because it's very simple and very fast. It's not just extremely simple to implement, it's also extremely simple to apply. Being a synchronous *stream cipher*, there's little that can go wrong; with a block cipher, you'd have to worry about things like modes of operation and padding. Clocking in at around 13.9 cycles per byte, it's comparable to AES-128 in CTR (12.6 cycles per byte) or CBC (16.0 cycles per byte) modes. AES came out a few years after RC4; when RC4 was designed, the state of the art was 3DES, which was excruciatingly slow by comparison (134.5 cycles per byte in *CTR mode*). [Dai]

## An in-depth look at RC4



This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

On the other hand, RC4 is incredibly simple, and it may be worth skimming this section.

RC4 is, unfortunately, quite broken. To better understand just how broken, we'll take a look at how RC4 works. The description requires understanding modular addition; if you aren't familiar with it, you may want to review *the appendix on modular addition* (page 186).

Everything in RC4 revolves around a state array and two indexes into that array. The array consists of 256 bytes forming a *permutation*: that is, all possible index values occur exactly once as a value in the array. That means it maps every possible byte value to every possible byte value: usually different, but sometimes the same one. We know that it's a permutation because $S$ starts as one, and all operations that modify $S$ always swap values, which obviously keeps it a permutation.

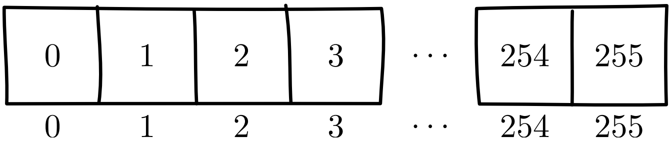
RC4 consists of two major components that work on two indexes $i, j$ and the state array $S$:

1. The key scheduling algorithm, which produces an initial state array $S$ for a given key.

2. The pseudorandom generator, which produces the actual keystream bytes from the state array $S$ which was produced by the key scheduling algorithm. The pseudorandom generator itself modifies the state array as it produces keystream bytes.

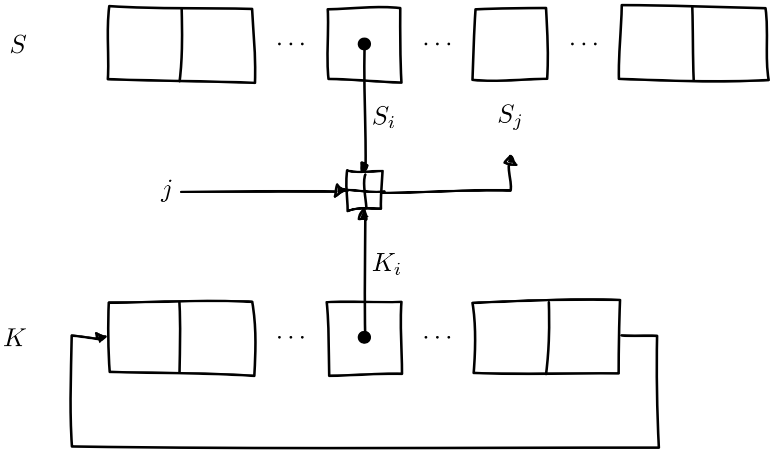
## The key scheduling algorithm

The key scheduling algorithm starts with the *identity permutation.* That means that each byte is mapped to itself.



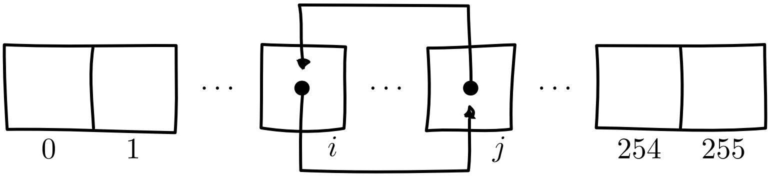
| 0 | 1 | 2 | 3 | $\cdots$ | 254 | 255 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | $\cdots$ | 254 | 255 |

Then, the key is mixed into the state. This is done by letting index $i$ iterate over every element of the state. The $j$ index is found by adding the current value of $j$ (starting at 0) with the next byte of the key, and the current state element:



Once $j$ has been found, $S[i]$ and $S[j]$ are swapped:

This process is repeated for all the elements of $S$. If you run out of key bytes, you just wrap around on the key. This explains why RC4 accepts keys from anywhere between 1



and 256 bytes long. Usually, 128 bit (16 byte) keys are used, which means that each byte in the key is used 16 times.

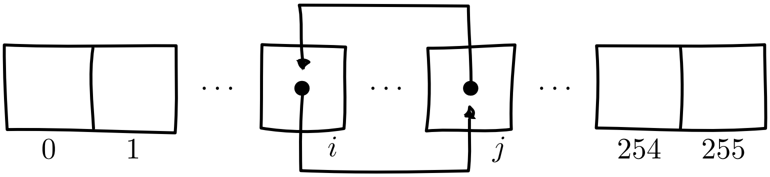
Or, in Python:

```
from itertools import import cycle
```

```
from itertools import cycle

def key_schedule(key):
    s = range(256)
    key_bytes = cycle(ord(x) for x in key)

    j = 0
    for i in range(256):
        j = (j + s[i] + next(key_bytes)) % 256
        s[i], s[j] = s[j], s[i]

    return s
```

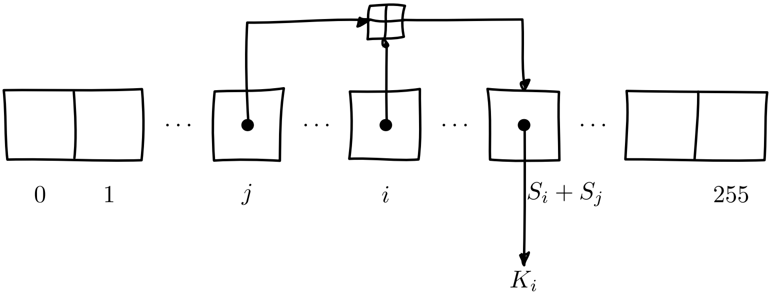
## The pseudorandom generator

The pseudorandom generator is responsible for producing pseudorandom bytes from the state $S$. These bytes form the keystream, and are XORed with the plaintext to produce the ciphertext. For each index $i$, it computes $j = j + S[i]$ ($j$ starts at 0). Then, $S[i]$ and $S[j]$ are swapped:



To produce the output byte, $S[i]$ and $S[j]$ are added together. Their sum is used as an index into $S$; the value at

$S[S[i] + S[j]]$ is the keystream byte $K_i$:



We can express this in Python:

```
def pseudorandom_generator(s):
    j = 0
    for i in cycle(range(256)):
        j = (j + s[i]) % 256
        s[i], s[j] = s[j], s[i]

        k = (s[i] + s[j]) % 256
        yield s[k]
```

## Attacks

This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

The section on the attacks on RC4 is a good deal more complicated than RC4 itself, so you may want to skip this even if you've read this far.

There are many attacks on RC4-using cryptosystems where RC4 isn't really the issue, but are caused by things like key reuse or failing to authenticate the message. We won't discuss these in this section. Right now, we're only talking about issues specific to the RC4 algorithm itself.

Intuitively, we can understand how an ideal *stream cipher* would produce a stream of random bits. After all, if that's what it did, we'd end up in a situation quite similar to that of a one-time pad.

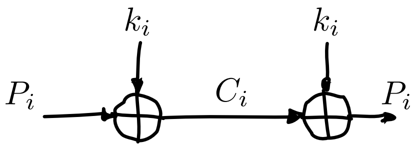


Figure 7.2: A one-time pad scheme.

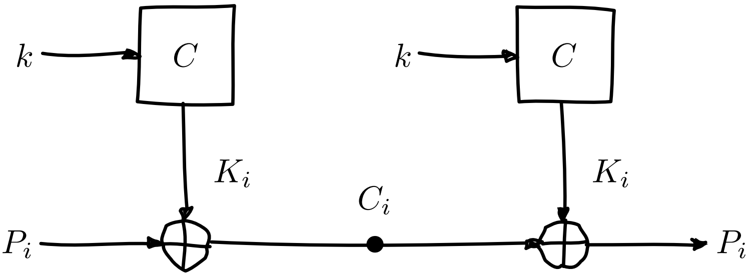


Figure 7.3: A synchronous *stream cipher* scheme. Note similarity to the one-time pad scheme. The critical difference is that while the one-time pad $k_i$ is truly random, the keystream $K_i$ is only pseudorandom.

The *stream cipher* is ideal if the best way we have to attack it is to try all of the keys, a process called brute-forcing the key. If there's an easier way, such as through a bias in the output bytes, that's a flaw of the *stream cipher*.

Throughout the history of RC4, people have found many such biases. In the mid-nineties, Andrew Roos noticed two such flaws:

- The first three bytes of the key are correlated with the first byte of the keystream.
- The first few bytes of the state are related to the key with a simple (linear) relation.

For an ideal *stream cipher*, the first byte of the keystream should tell me nothing about the key. In RC4, it gives me some information about the first three bytes of the key. The latter seems less serious: after all, the attacker isn't supposed to know the state of the cipher.

As always, attacks never get worse. They only get better.

Adi Shamir and Itsik Mantin showed that the second byte produced by the cipher is *twice* as likely to be zero as it should be. Other researchers showed similar biases in the first few bytes of the keystream. This sparked further research by Mantin, Shamir and Fluhrer, showing large biases in the first bytes of the keystream. [FMS01] They also showed that knowing even small parts of the key would allow attackers to make strong predictions about the state and outputs of the cipher. Unlike RC4, most modern stream ciphers provide a way to combine a long-term key with a *nonce* (a number used once), to produce multiple different keystreams from the same long-term key. RC4, by itself, doesn't do that. The most common approach was also the simplest: concatenate[6] the long-term key $k$ with the *nonce* $n$: $k \| n$, taking advantage of RC4's flexible key length requirements. In this context, concatenation means the bits of $n$ are appended to the bits of $k$. This scheme meant attackers could recover parts of the combined key, eventually allowing them to slowly recover the long-term key from a large amount of messages (around $2^{24}$ to $2^{26}$, or tens of millions of messages).

WEP, a standard for protecting wireless networks that was popular at the time, was heavily affected by this attack, because it used this simplistic *nonce* combination scheme. A scheme where the long-term key and the *nonce* had been securely combined (for example using a key derivation function or a cryptographic hash function) wouldn't have had

[6] Here we use $\|$ as the operator for concatenation. Other common symbols for concatenation include $+$ (for some programming languages, such as Python) and ⊠ (for formal languages).

this weakness. Many other standards including TLS were therefore not affected.

Again, attacks only get better. Andreas Klein showed more extensive correlation between the key and the keystream. [Kle08] Instead of tens of millions of messages with the Fluhrer, Mantin, Shamir attacks, attackers now only needed several tens of thousands of messages to make the attack practical. This was applied against WEP with great effect.

In 2013, a team of researchers at Royal Holloway in London produced a combination of two independent practical attacks [ABP+]. These attacks proved to be very damning for RC4: while RC4's weaknesses had been known for a long time, they finally drove the point home for everyone that it really shouldn't be used anymore.

The first attack is based on single-byte biases in the first 256 bytes of the keystream. By performing statistical analysis on the keystreams produced by a large number of keys, they were able to analyze the already well-known biases in the early keystream bytes of RC4 in much greater detail.

TODO: illustrate: `http://www.isg.rhul.ac.uk/tls/RC4_keystream_dist_2_45.txt`

The second attack is based on double byte biases anywhere in the keystream. It turns out that adjacent bytes of the keystream have an exploitable relation, whereas in an ideal *stream cipher* you would expect them to be completely independent.

| Byte pair | Byte position (mod 256) $i$ | Probability |
|---|---|---|
| $(0, 0)$ | $i = 1$ | $2^{-16}(1 + 2^{-9})$ |
| $(0, 0)$ | $i \notin \{1, 255\}$ | $2^{-16}(1 + 2^{-8})$ |
| $(0, 1)$ | $i \notin \{0, 1\}$ | $2^{-16}(1 + 2^{-8})$ |
| $(0, i + 1)$ | $i \notin \{0, 255\}$ | $2^{-16}(1 + 2^{-8})$ |
| $(i + 1, 255)$ | $i \neq 254$ | $2^{-16}(1 + 2^{-8})$ |
| $(255, i + 1)$ | $i \notin \{1, 254\}$ | $2^{-16}(1 + 2^{-8})$ |
| $(255, i + 2)$ | $i \notin \{0, 253, 254, 255\}$ | $2^{-16}(1 + 2^{-8})$ |
| $(255, 0)$ | $i = 254$ | $2^{-16}(1 + 2^{-8})$ |
| $(255, 1)$ | $i = 255$ | $2^{-16}(1 + 2^{-8})$ |
| $(255, 2)$ | $i \in \{0, 1\}$ | $2^{-16}(1 + 2^{-8})$ |
| $(255, 255)$ | $i \neq 254$ | $2^{-16}(1 + 2^{-8})$ |
| $(129, 129)$ | $i = 2$ | $2^{-16}(1 + 2^{-8})$ |

This table may seem a bit daunting at first. The probability expression in the rightmost column may look a bit complex, but there's a reason it's expressed that way. Suppose that RC4 was a good *stream cipher*, and all values occurred with equal probability. Then you'd expect the probability for any given byte value to be $2^{-8}$ since there are $2^8$ different byte values. If RC4 was a good *stream cipher*, two adjacent bytes would each have probability $2^{-8}$, so any given pair of two bytes would have probability $2^{-8} \cdot 2^{-8} = 2^{-16}$. However, RC4 isn't an ideal *stream cipher*, so these properties aren't true. By writing the probability in the $2^{-16}(1 + 2^{-k})$ form, it's easier to see how much RC4 deviates from what you'd expect from an ideal *stream cipher*.

So, let's try to read the first line of the table. It says that when the first byte $i = 1$ of any 256-byte chunk from the cipher is $0$, then the byte following it is slightly more likely ($1 + 2^{-9}$ times as likely, to be exact) to be 0 than for it to be any other number. We can also see that when one of the keystream bytes is $255$, you can make many predictions about the next byte, depending on where it occurs in the keystream. It's more likely to be $0, 1, 2, 255$, or the position in the keystream plus one or two.

TODO: demonstrate attack success

Again, attacks only get better. These attacks have primarily focused on the cipher itself, and haven't been fully optimized for practical attacks on, say, web services. The attacks can be greatly improved with some extra information about the plaintext you're attempting to recover. For example, HTTP cookies are often base-64 or hex encoded.

There's no way around it: we need to stop using RC4. Fortunately, we've also developed many secure alternatives. The continuing advances in cryptanalysis of RC4 helped contribute to a sense of urgency regarding the improvement of commonly available cryptographic primitives. Throughout 2013 in particular, this led to large improvements in, for example, browser cryptography (we will discuss browser cryptography, notably SSL/TLS, in a later chapter).

## 7.12 Salsa20

Salsa20 is a newer *stream cipher* designed by Dan Bernstein. Bernstein is well-known for writing a lot of open source (public domain) software, most of which is either directly security related or built with information security very much in mind.

There are two minor variants of Salsa20, called Salsa20/12 and Salsa20/8, which are simply the same algorithm except with 12 and 8 rounds[7] respectively, down from the original 20. ChaCha is another, orthogonal tweak of the Salsa20 cipher, which tries to increase the amount of diffusion per round while maintaining or improving performance. ChaCha doesn't have a "20" after it; specific algorithms do have a number after them (ChaCha8, ChaCha12, ChaCha20), which refers to the number of rounds.

[7] Rounds are repetitions of an internal function. Typically a number of rounds are required to make an algorithm work effectively; attacks often start on reduced-round versions of an algorithm.

Salsa20 and ChaCha are among the state of the art of modern stream ciphers. There are currently no publicly known attacks against Salsa20, ChaCha, nor against any of their recommended reduced-round variants, that break their practical security.

Both cipher families are also pretty fast. For long streams, Salsa20 takes about 4 cycles per byte for the full-round version, about 3 cycles per byte for the 12-round version and about 2 cycles per byte for the 8-round version, on modern Intel processors [Ber] and modern AMD processors [Dai]. ChaCha is (on most platforms) slightly faster still. To put that into comparison, that's more than three times faster than RC4[8], approximately three times faster than AES-CTR with a 128 bit key at 12.6 cycles per byte, and roughly in the ballpark of AES *GCM mode*[9] with specialized hardware instructions.

Salsa20 has two particularly interesting properties. Firstly, it is possible to "jump" to a particular point in the keystream without computing all previous bits. This can be useful, for example, if a large file is encrypted, and you'd like to be able to do random reads in the middle of the file. While many encryption schemes require the entire file to be decrypted, with Salsa20, you can just select the portion you need. Another construction that has this property is a *mode of operation* called *CTR mode*, which we'll talk about later.

This ability to "jump" also means that blocks from Salsa20 can be computed independently of one another, allowing for encryption or decryption to work in parallel, which can increase performance on multi-core CPUs.

Secondly, it is resistant to many side-channel attacks. This is done by ensuring that no key material is ever used

[8] The quoted benchmarks don't mention RC4 but MARC4, which stands for "modified alleged RC4". The RC4 section explains why it's "alleged", and "modified" means it throws away the first 256 bytes because of a weakness in RC4.

[9] *GCM mode* is an authenticated encryption mode, which we will see in more detail in a later chapter.

to choose between different code paths in the cipher, and that every round is made up of a fixed-number of constant-time operations. The result is that every block is produced with exactly the same number of operations, regardless of what the key is.

Both *stream ciphers* are based on an ARX design. One benefit of ARX ciphers is that they are intrinsically constant time. There are no secret memory access patterns that might leak information, as with AES. These ciphers also perform well on modern CPU architectures without needing cipher-specific optimizations. They take advantage of generic vector instructions, where the CPU performs related operations on multiple pieces of data in a single instruction. As a result, ChaCha20 performance is competitive with AES on modern Intel CPUs, even though the latter has specialized hardware.

Here is an example ARX operation:

$$x \leftarrow x \oplus (y \boxplus z) \lll n$$

To find the new value of $x$, first we perform a modular addition ($\boxplus$) of $y$ and $z$, then we XOR ($\oplus$) the result with x and finally we rotate left ($\lll$) by $n$ bits. This is the core round primitive of Salsa20.

## 7.13 Native stream ciphers versus modes of operation

Some texts only consider native *stream ciphers* to be *stream ciphers*. This book emphasizes what the functionality of the algorithm is. Since both block ciphers in a *mode of operation* and a native *stream cipher* take a secret key and can be used to encrypt a stream, and the two can usually replace each other in a cryptosystem, we just call both of them *stream ciphers* and are done with it.

We will further emphasize the tight link between the two with *CTR mode*, a *mode of operation* which produces a synchronous *stream cipher*. While there are also modes of operation (like OFB and CFB) that can produce self-synchronizing *stream cipher*s, these are far less common, and not discussed here.

## 7.14 CTR mode

*CTR mode*, short for counter mode, is a *mode of operation* that works by concatenating a *nonce* with a counter. The counter is incremented with each block, and padded with zeroes so that the whole is as long as the block size. The resulting concatenated string is run through a block cipher. The outputs of the block cipher are then used as the keystream.

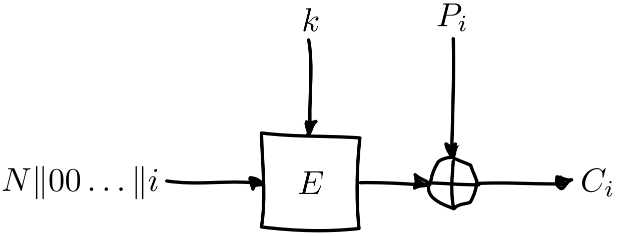


Figure 7.4: *CTR mode*: a single *nonce* $N$ with a zero-padded counter $i$ is encrypted by the block cipher to produce a keystream block; this block is XORed with the plaintext block $P_i$ to produce the ciphertext block $C_i$.

This illustration shows a single input block $N\|00\ldots\|i$, consisting of *nonce* $N$, current counter value $i$ and padding, being encrypted by the block cipher $E$ using key $k$ to produce keystream block $S_i$, which is then XORed with the plaintext block $P_i$ to produce ciphertext block $C_i$.

Obviously, to decrypt, you do the exact same thing again, since XORing a bit with the same value twice always produces the original bit: $p_i \oplus s_i \oplus s_i = p_i$. As a consequence, CTR encryption and decryption is the same thing: in both cases you produce the keystream, and you XOR either the plaintext or the ciphertext with it in order to get the other one.

For *CTR mode* to be secure, it is critical that *nonce*s aren't reused. If they are, the entire keystream will be repeated, allowing an attacker to mount multi-time pad attacks.

This is different from an *initialization vector* such as the one used by CBC. An IV has to be unpredictable. An attacker being able to predict a CTR *nonce* doesn't really matter: without the secret key, they have no idea what the output of the block cipher (the sequence in the keystream) would be.

Like Salsa20, *CTR mode* has the interesting property that you can jump to any point in the keystream easily: just increment the counter to that point. *The Salsa20 paragraph on this topic* (page 76) explains why that might be useful.

Another interesting property is that since any keystream block can be computed completely separately from any other keystream block, both encryption and decryption are very easy to compute in parallel.

## 7.15 Stream cipher bit flipping attacks

Synchronous *stream cipher*s, such as native *stream cipher*s or a block cipher in *CTR mode*, are also vulnerable to a bit flipping attack. It's similar to CBC bit flipping attacks in the sense that an attacker flips several bits in the ciphertext, and that causes some bits to be flipped in the plaintext.

This attack is actually much simpler to perform on *stream cipher*s than it is on *CBC mode*. First of all, a flipped bit in the ciphertext results in the same bit being flipped in the plaintext, not the corresponding bit in the following block. Additionally, it only affects that bit; in CBC bit flipping attacks, the plaintext of the modified block is scrambled. Finally, since the attacker is modifying a sequence of bytes and not a sequence of blocks, the attacks are not limited by the specific block size. In CBC bit flipping attacks, for example, an attacker can adjust a single block, but can't adjust the adjacent block.

TODO illustrate

This is yet another example of why authentication has to go hand in hand with encryption. If the message is properly authenticated, the recipient can simply reject the modified messages, and the attack is foiled.

## 7.16 Authenticating modes of operation

There are other modes of operation that provide authentication as well as encryption at the same time. Since we haven't discussed authentication at all yet, we'll handle these later.

## 7.17 Remaining problems

We now have tools that will encrypt large streams of data using a small key. However, we haven't actually discussed how we're going to agree on that key. As noted in a previous chapter, to communicate between $n$ people, we need $\frac{n(n-1)}{2}$ key exchanges. The number of key exchanges grows about as fast as the number of people *squared*. While the key to be exchanged is a lot smaller now than it was with one-time pads, the fundamental problem of the impossibly large number of key exchanges hasn't been solved yet. We will tackle that problem in the next section, where we'll look at key exchange protocols: protocols that allow us to agree on a secret key over an insecure medium.

Additionally, we've seen that encryption isn't enough to provide security: without authentication, it's easy for attackers to modify the message, and in many flawed systems even decrypt messages. In a future chapter, we'll discuss how to *authenticate* messages, to prevent attackers from modifying them.

# 8

# Key exchange

## 8.1 Description

*Key exchange* protocols attempt to solve a problem that, at first glance, seems impossible. Alice and Bob, who've never met before, have to agree on a secret value. The channel they use to communicate is insecure: we're assuming that everything they send across the channel is being eavesdropped on.

We'll demonstrate such a protocol here. Alice and Bob will end up having a shared secret, only communicating over the insecure channel. Despite Eve having literally all of the information Alice and Bob send to each other, she can't use any of that information to figure out their shared secret.

That protocol is called Diffie-Hellman, named after Whitfield Diffie and Martin Hellman, the two cryptographic pioneers who discovered it. They suggested calling the protocol Diffie-Hellman-Merkle *key exchange*, to honor the contributions of Ralph Merkle. While his contributions certainly deserve honoring, that term hasn't really caught on. For the benefit of the reader we'll use the more common term.

Practical implementations of Diffie-Hellman rely on mathematical problems that are believed to be very complex to solve in the "wrong" direction, but easy to compute in the "right" direction. Understanding the mathematical implementation isn't necessary to understand the principle behind the protocol. Most people also find it a lot easier to understand without the mathematical complexity. So, we'll explain Diffie-Hellman in the abstract first, without any mathematical constructs. Afterwards, we'll look at two practical implementations.

## 8.2 Abstract Diffie-Hellman

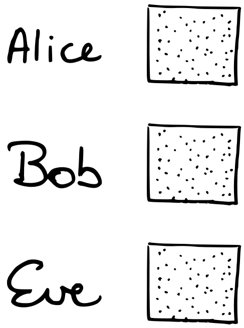
In order to describe Diffie-Hellman, we'll use an analogy based on mixing colors. We can mix colors according to the following rules:

- It's very easy to mix two colors into a third color.
- Mixing two or more colors in different order results in the same color.
- Mixing colors is *one-way*. It's impossible to determine if, let alone which, multiple colors were used to produce a given color. Even if you know it was mixed, and even if you know some of the colors used to produce it, you have no idea what the remaining color(s) were.

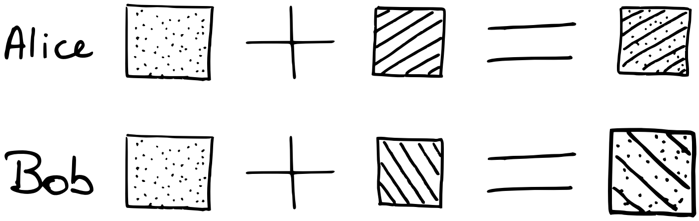
We'll demonstrate that with a mixing function like this one, we can produce a secret color only known by Alice and Bob. Later, we'll simply have to describe the concrete implementation of those functions to get a concrete *key exchange* scheme.

To illustrate why this remains secure in the face of eavesdroppers, we'll walk through an entire exchange with Eve, the eavesdropper, in the middle. Eve is listening to all of the messages sent across the network. We'll keep track of everything she knows and what she can compute, and end up seeing *why* Eve can't compute Alice and Bob's shared secret.

To start the protocol, Alice and Bob have to agree on a base color. They can communicate that across the network: it's okay if Eve intercepts the message and finds out what the color is. Typically, this base color is a fixed part of the protocol; Alice and Bob don't need to communicate it. After this step, Alice, Bob and Eve all have the same information: the base color.

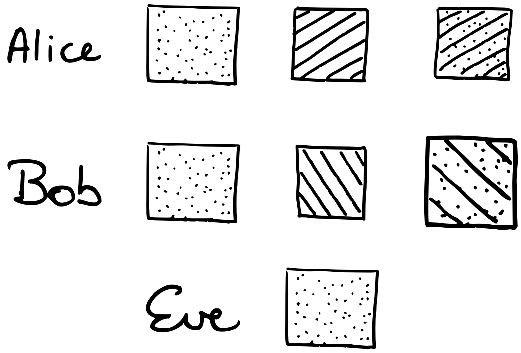


Alice and Bob both pick a random color, and they mix it with the base color.

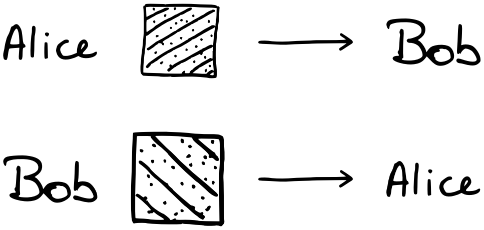


At the end of this step, Alice and Bob know their respective secret color, the mix of the secret color and the base color, and the base color itself. Everyone, including Eve, knows the base color.

Then, Alice and Bob both send their mixed colors over the network. Eve sees both mixed colors, but she can't fig-



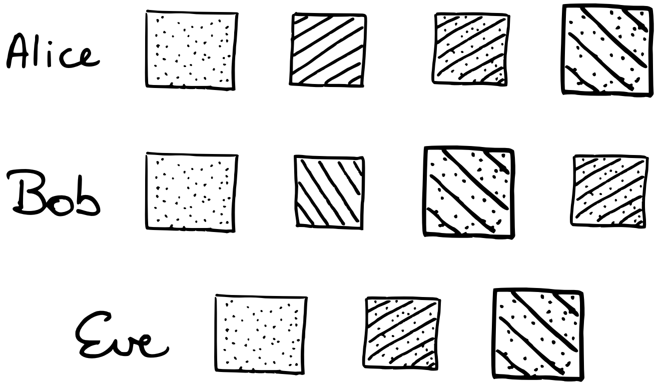
ure out what either of Alice and Bob's *secret* colors are. Even though she knows the base, she can't "un-mix" the colors sent over the network.[1]

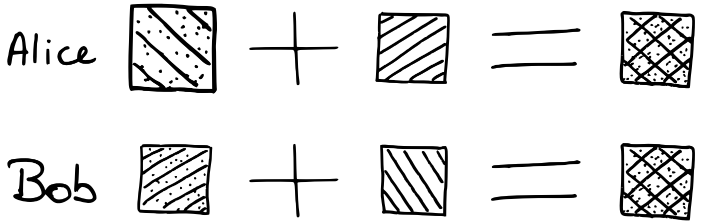


At the end of this step, Alice and Bob know the base, their respective secrets, their respective mixed colors, and each other's mixed colors. Eve knows the base color and both mixed colors.

Once Alice and Bob receive each other's mixed color, they add their own secret color to it. Since the order of the mixing doesn't matter, they'll both end up with the same secret.

[1] While this might seem like an easy operation with black-and-white approximations of color mixing, keep in mind that this is just a failure of the illustration: our assumption was that this was hard.





Eve can't perform that computation. She could finish the computation with either Alice or Bob's secret color, since she has both mixed colors, but she has neither of those secret colors. She can also try to mix the two mixed colors, which would have both Alice and Bob's secret colors mixed into them. However, that would have the base color in it twice, resulting in a different color than the shared secret color that Alice and Bob computed, which only has the base color in it once.

## 8.3 Diffie-Hellman with discrete logarithms

This section describes a practical implementation of the Diffie-Hellman algorithm, based on the discrete logarithm problem. It is intended to provide some mathematical background, and requires modular arithmetic to understand. If you are unfamiliar with modular arithmetic, you can either skip this chapter, or first read the *mathematical background appendix* (page 186).

Discrete log Diffie-Hellman is based on the idea that computing $y$ in the following equation is easy (at least for a computer):

$$y \equiv g^x \pmod{p}$$

However, computing $x$ given $y$, $g$ and $p$ is believed to be very hard. This is called the discrete logarithm problem, because a similar operation without the modular arithmetic is called a logarithm.

This is just a concrete implementation of the abstract Diffie-Hellman process we discussed earlier. The common base color is a large prime $p$ and the base $g$. The "color mixing" operation is the equation given above, where $x$ is the input value and $y$ is the resulting mixed value.

When Alice or Bob select their random numbers $r_A$ and $r_B$, they mix them with the base to produce the mixed numbers $m_A$ and $m_B$:

$$m_A \equiv g^{r_A} \pmod{p}$$

$$m_B \equiv g^{r_B} \pmod{p}$$

These numbers are sent across the network where Eve can see them. The premise of the discrete logarithm problem is that it is okay to do so, because figuring out $r$ in $m \equiv g^r \pmod{p}$ is supposedly very hard.

Once Alice and Bob have each other's mixed numbers, they add their own secret number to it. For example, Bob would compute:

$$s \equiv (g^{r_A})^{r_B} \pmod{p}$$

While Alice's computation looks different, they get the same result, because $(g^{r_A})^{r_B} \equiv (g^{r_B})^{r_A} \pmod{p}$. This is the shared secret.

Because Eve doesn't have $r_A$ or $r_B$, she can not perform the equivalent computation: she only has the base number $g$ and mixed numbers $m_A \equiv g^{r_A} \pmod{p}$ and $m_B \equiv g^{r_B} \pmod{p}$ , which are useless to her. She needs either $r_A$ or $r_B$ (or both) to make the computation Alice and Bob do.

TODO: Say something about active MITM attacks where the attacker picks smooth values to produce weak secrets?

## 8.4 Diffie-Hellman with elliptic curves

This section describes a practical implementation of the Diffie-Hellman algorithm, based on the elliptic curve discrete logarithm problem. It is intended to provide some mathematical background, and requires a (very basic) understanding of the mathematics behind elliptic curve cryptography. If you are unfamiliar with elliptic curves, you can either skip this chapter, or first read the *mathematical background appendix* (page 202).

One of the benefits of the elliptic curve Diffie-Hellman variant is that the required key size is much, much smaller than the variant based on the discrete log problem. This is because the fastest algorithms for breaking the discrete log problem have a larger asymptotic complexity than their elliptic curve variants. For example, the number field sieve for discrete logarithms, a state of the art algorithm for attacking discrete logarithm-based Diffie-Hellman, has time complexity:

$$L\left[1/3, \sqrt[3]{64/9}\right]$$

Which is more than polynomial (but less than exponential) in the number of digits. On the other hand, the fastest algorithms that could be used to break the elliptic curve discrete log problem all have complexity:

$$L\left[1, 1/2\right] = O(\sqrt{n})$$

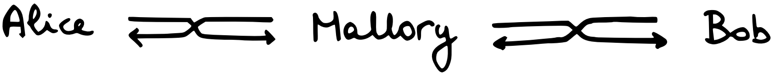
Relatively speaking, that means that it's much harder to solve the elliptic curve problem than it is to solve the regular discrete log problem, using state of the art algorithms for both. The flip side of that is that for equivalent security levels, the elliptic curve algorithm needs much smaller key sizes [Lab] [InstitutefStandardsTechnology][2]:

| Security level in bits | Discrete log key bits | Elliptic curve key bits |
|---|---|---|
| 56 | 512 | 112 |
| 80 | 1024 | 160 |
| 112 | 2048 | 224 |
| 128 | 3072 | 256 |
| 256 | 15360 | 512 |

## 8.5 Remaining problems

Using Diffie-Hellman, we can agree on shared secrets across an insecure Internet, safe from eavesdroppers. However, while an attacker may not be able to simply get the secret from eavesdropping, an active attacker can still break the system. If such an attacker, usually called Mallory, is in between Alice and Bob, she can still perform the Diffie-Hellman protocol twice: once with Alice, where Mallory pretends to be Bob, and once with Bob, where Mallory pretends to be Alice.

[2] These figures are actually for the RSA problem versus the equivalent elliptic curve problem, but their security levels are sufficiently close to give you an idea.



There are two shared secrets here: one between Alice and Mallory, and one between Mallory and Bob. The attacker (Mallory) can then simply take all the messages they get from one person and send them to the other, they can look at the plaintext messages, remove messages, and they can also modify them in any way they choose.

To make matters worse, even if one of the two participants was somehow aware that this was going on, they would have no way to get the other party to believe them. After all: Mallory performed the successful Diffie-Hellman exchange with the unwitting victim, she has all the correct shared secrets. Bob has no shared secrets with Alice, just with Mallory; there's no way for him to prove that he's the legitimate participant. As far as Alice can tell, Bob just chose a few random numbers. There's no way to link any key that Bob has with any key that Alice has.

Attacks like these are called MITM attacks, because the attacker (Mallory) is in between the two peers (Alice and Bob). Given that the network infrastructure that we typically use to send messages is run by many different operators, this kind of attack scenario is very realistic, and a secure cryptosystem will have to address them somehow.

While the Diffie-Hellman protocol successfully produced a shared secret between two peers, there are clearly some pieces of the puzzle still missing to build those cryptosystems. We need tools that help us authenticate Alice to Bob and vice versa, and we need tools that help guarantee message integrity, allowing the receiver to verify that the received messages are in fact the messages the sender intended to send.

9

# Public-key encryption

## 9.1 Description

So far, we have only done *secret-key encryption*. Suppose, that you could have a cryptosystem that didn't involve a single secret key, but instead had a key pair: one public key, which you freely distribute, and a private one, which you keep to yourself.

People can encrypt information intended for you by using your public key. The information is then impossible to decipher without your private key. This is called *public-key encryption*.

For a long time, people thought this was impossible. However, starting in the 1970s, such algorithms started appearing. The first publicly available encryption scheme was produced by three cryptographers from MIT: Ron Rivest, Adi Shamir and Leonard Adleman. The algorithm they published is still the most common one today, and carries the first letters of their last names: RSA.

*public-key algorithm*s aren't limited to encryption. In fact, you've already seen a *public-key algorithm* in this book that isn't directly used for encryption. There are actually three related classes of *public-key algorithms*:

1. *Key exchange* algorithms, such as Diffie-Hellman, which allow you to agree on a shared secret across an insecure medium.
2. Encryption algorithms, such as the ones we'll discuss in this chapter, which allow people to encrypt without having to agree on a shared secret.
3. Signature algorithms, which we'll discuss in a later chapter, which allow you to sign any piece of information using your private key in a way that allows anyone else to easily verify it using your public key.

## 9.2 Why not use public-key encryption for everything?

At face value, it seems that *public-key encryption* algorithms obsolete all our previous *secret-key encryption* algorithms. We could just use public key encryption for everything, avoiding all the added complexity of having to do *key agreement* for our symmetric algorithms. However, when we look at practical cryptosystems, we see that they're almost always *hybrid* cryptosystems: while *public-key algorithms* play a very important role, the bulk of the encryption and authentication work is done by secret-key algorithms.

By far the most important reason for this is performance. Compared to our speedy *stream ciphers* (native or otherwise), *public-key encryption* mechanisms are extremely slow. RSA is limited to at most its key size, which for 2048-bit means 256 bytes. Under these circumstances encryption takes 0.29 megacycles, and decryption takes a whopping 11.12 megacycles. [Dai] To put this into perspective, symmetric key algorithms work within an order of magnitude of 10 or so cycles per byte in either direction. This means it will take a symmetric key algorithm approximately 3 kilocycles in order to decrypt 256 bytes, which is about 4000 times faster than the asymmetric version. The state of the art in secure symmetric ciphers is even faster: AES-GCM with hardware acceleration or Salsa20/ChaCha20 only need about 2 to 4 cycles per byte, further widening the performance gap.

There are a few other problems with most practical cryptosystems. For example, RSA can't encrypt anything larger than its modulus, which is generally less than or equal 4096 bits, far smaller than the largest messages we'd like to send. Still, the most important reason is the speed argument given above.

## 9.3 RSA

As we already mentioned, RSA is one of the first practical *public-key encryption* schemes. It remains the most common one to this day.

### Encryption and decryption

RSA encryption and decryption relies on modular arithmetic. You may want to review the *modular arithmetic primer* (page 186) before continuing.

This section describes the simplified math problem behind RSA, commonly referred to as "textbook RSA". By itself, this doesn't produce a secure encryption scheme. We'll see a secure construction called OAEP that builds on top of it in a later section.

In order to generate a key, you pick two large prime numbers $p$ and $q$. These numbers have to be picked at random, and in secret. You multiply them together to produce the modulus $N$, which is public. Then, you pick an *encryption exponent* $e$, which is also public. Usually, this value is either 3 or 65537. Because those numbers have a small number of 1's in their binary expansion, you can compute the exponentiation more efficiently. Put together, $(N, e)$ is the public key. Anyone can use the public key to encrypt a message $M$ into a ciphertext $C$:

$$C \equiv M^e \pmod{N}$$

The next problem is decryption. It turns out that there is a value $d$, the *decryption exponent*, that can turn $C$ back into $M$. That value is fairly easy to compute assuming that you know $p$ and $q$, which we do. Using $d$, you can decrypt the message like so:

$$M \equiv C^d \pmod{N}$$

The security of RSA relies on that decryption operation being impossible without knowing the secret exponent $d$, and that the secret exponent $d$ is very hard (practically impossible) to compute from the public key $(N, e)$. We'll see approaches for breaking RSA in the next section.

## Breaking RSA

Like many cryptosystems, RSA relies on the presumed difficulty of a particular mathematical problem. For RSA, this is the RSA problem, specifically: to find the plaintext message $M$, given a ciphertext $C$, and public key $(N, e)$ in the equation:

$$C \equiv M^e \pmod{N}$$

The easiest way we know how to do that is to factor $N$ back into $p \cdot q$. Given $p$ and $q$, the attacker can just repeat the process that the legitimate owner of the key does during key generation in order to compute the private exponent $d$.

Fortunately, we don't have an algorithm that can factor such large numbers in reasonable time. Unfortunately, we also haven't proven it doesn't exist. Even more unfortunate is that there is a theoretical algorithm, called Shor's algorithm, that *would* be able to factor such a number in reasonable time on a quantum computer. Right now, quantum computers are far from practical, but it does appear that if someone in the future manages to build one that's sufficiently large, RSA becomes ineffective.

In this section, we have only considered a private key recovery attack that attacks the purely abstract mathematical RSA problem by factoring the modulus. In the next section, we will see all sorts of realistic attacks on RSA that rely on flaws in the *implementation*, rather than the mathematical problem stated above.

## Implementation pitfalls

Right now, there are no known practical complete breaks against RSA. That's not to say that systems employing RSA aren't routinely broken. Like with most broken cryptosystems, there are plenty of cases where sound components, improperly applied, result in a useless system. For a more complete overview of the things that can go wrong with RSA implementations, please refer to [Bon99] and [AV96]. In this book, we'll just highlight a few interesting ones.

### PKCSv1.5 padding

### Salt

Salt[1] is a provisioning system written in Python. It has one major flaw: it has a module named `crypt`. Instead of reusing existing complete cryptosystems, it implements its own, using RSA and AES provided by a third party package.

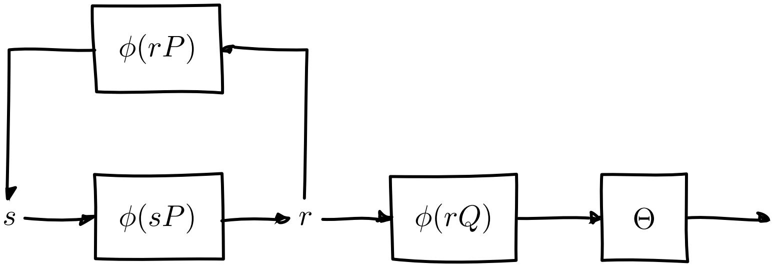
For a long time, Salt used a public exponent ($e$) of 1, which meant the encryption phase didn't actually do anything: $P^e \equiv P^1 \equiv P \pmod{N}$. This meant that the resulting ciphertext was in fact just the plaintext. While this issue has now been fixed, this only goes to show that you probably

[1] So, there's Salt the provisioning system, *salt*s the things used in broken password stores, NaCl pronounced “salt” the cryptography library, and NaCl which runs native code in some browsers, and probably a bunch I'm forgetting. Can we stop naming things after it?

shouldn't implement your own cryptography. Salt currently also supports SSH as a transport, but the aforementioned DIY RSA/AES system remains, and is at time of writing still the recommended and the default transport.

## OAEP

OAEP, short for optimal asymmetric encryption padding, is the state of the art in RSA padding. It was introduced by Mihir Bellare and Phillip Rogaway in 1995. [BR95]. Its structure looks like this:



The thing that eventually gets encrypted is $X \| Y$, which is $n$ bits long, where $n$ is the number of bits of $N$, the RSA modulus. It takes a random block $R$ that's $k$ bits long, where $k$ is a constant specified by the standard. The message is first padded with zeroes to be $n - k$ bits long. If you look at the above "ladder", everything on the left half is $n - k$ bits long, and everything on the right half is $k$ bits long. The random block $R$ and zero-padded message $M \| 000 \ldots$ are combined using two "trapdoor" functions, $G$ and $H$. A trapdoor function is a function that's very easy to compute in one direction and very hard to reverse. In practice, these are cryptographic hash functions; we'll see more about those later.

As you can tell from the diagram, $G$ takes $k$ bits and turns them into $n - k$ bits, and $H$ is the other way around, taking $n - k$ bits and turning them into $k$ bits.

The resulting blocks $X$ and $Y$ are concatenated, and the result is encrypted using the standard RSA encryption primitive, to produce the ciphertext.

To see how decryption works, we reverse all the steps. The recipient gets $X\|Y$ when decrypting the message. They know $k$, since it is a fixed parameter of the protocol, so they can split up $X\|Y$ into $X$ (the first $n-k$ bits) and $Y$ (the final $k$ bits).

In the previous diagram, the directions are for padding being applied. Reverse the arrows on the side of the ladder, and you can see how to revert the padding:

TODO: reverse arrows

We want to get to $M$, which is in $M\|000\ldots$. There's only one way to compute that, which is:

$$M\|000\ldots = X \oplus G(R)$$

Computing $G(R)$ is a little harder:

$$G(R) = G(H(X) \oplus Y)$$

As you can see, at least for some definitions of the functions $H$ and $G$, we need all of $X$ and all of $Y$ (and hence the entire encrypted message) in order to learn anything about $M$. There are many functions that would be a good choice for $H$ and $G$; based on cryptographic hash functions, which we'll discuss in more detail later in the book.

## 9.4 Elliptic curve cryptography

TODO: This

## 9.5 Remaining problem: unauthenticated encryption

Most *public-key encryption* schemes can only encrypt small chunks of data at a time, much smaller than the messages we want to be able to send. They are also generally quite slow, much slower than their symmetric counterparts. Therefore public-key cryptosystems are almost always used in conjunction with secret-key cryptosystems.

When we discussed *stream cipher*s, one of the remaining issues that we were facing was that we still had to exchange secret keys with a large number of people. With public-key cryptosystems such as public encryption and *key exchange* protocols, we've now seen two ways that we can solve that problem. That means that we can now communicate with anyone, using only public information, completely secure from eavesdroppers.

So far we've only been talking about encryption without any form of authentication. That means that while we can encrypt and decrypt messages, we cannot verify that the message is what the sender actually sent.

While unauthenticated encryption may provide secrecy, we have already seen that without authentication an active attacker can generally modify valid encrypted messages successfully, despite the fact that they don't necessarily know the corresponding plaintext. Accepting these messages can often lead to secret information being leaked, meaning we don't even get secrecy. The CBC padding attacks we've already discussed illustrate this.

As a result it has become evident that we need ways to authenticate as well as encrypt our secret communications. This is done by adding extra information to the message that only the sender could have computed. Just like encryption, authentication comes in both private-key (symmetric) and public-key (asymmetric) forms. Symmetric authentication schemes are typically called *message authentication code*s, while the public-key equivalent is typically called a signature.

First, we will introduce a new cryptographic primitive: hash functions. These can be used to produce both signature schemes as well as message authentication schemes. Unfortunately, they are also very often abused to produce entirely insecure systems.

# 10

# Hash functions

## 10.1 Description

Hash functions are functions that take an input of indeterminate length and produce a fixed-length value, also known as a “digest”.

Simple hash functions have many applications. Hash tables, a common data structure, rely on them. These simple hash functions really only guarantee one thing: for two identical inputs, they’ll produce an identical output. Importantly, there’s no guarantee that two identical outputs imply that the inputs were the same. That would be impossible: there’s only a finite amount of digests, since they’re fixed size, but there’s an infinite amount of inputs. A good hash function is also quick to compute.

Since this is a book on cryptography, we’re particularly interested in *cryptographic* hash functions. Cryptographic hash functions can be used to build secure (symmetric) message authentication algorithms, (asymmetric) signature algorithms, and various other tools such as random number generators. We’ll see some of these systems in detail in future chapters.

Cryptographic hash functions have much stronger properties than regular hash functions, such as one that you might find in a hash table. For a cryptographic hash function, we want it to be impossibly hard to:

1. modify a message without changing the hash.

2. generate a message that has a given hash.

3. find two different messages with the same hash.

The first property implies that cryptographic hash functions will exhibit something known as the “avalanche effect”. Changing even a single bit in the input will produce an avalanche of changes through the entire digest: each bit of the digest will have approximately 50% chance of flipping. That doesn’t mean that every change *will* cause approximately half of the bits to flip, but the cryptographic hash function does guarantee that the odds of that happening are extremely large. More importantly it is impossibly hard to find such collisions or near-collisions.

The second property, which states that it should be difficult to find a message $m$ that has a given hash value $h$, is called *pre-image resistance*. This makes a hash function a one-way function: it’s very easy to compute a hash for a given message, but it’s very hard to compute a message for a given hash.

The third property talks about finding messages with the same hash value, comes in two flavors. In the first one, there’s a given message $m$, and it should be difficult to find another message $m'$ with the same hash value: that’s called *second pre-image resistance*. The second one is stronger, stating that it should be hard to find any two messages $m, m'$ that have the same hash value. This is called *collision resistance*. Because collision resistance is a stronger form of second pre-image resistance, they’re sometimes also called weak and strong collision resistance.

These concepts are often named from the point of view of an attack, rather than the resistance to an attack. For example, you'll often hear about a collision attack, which is an attack that attempts to generate a hash collision, or a second pre-image attack, which attempts to find a second pre-image that hashes to the same value as a given pre-image, et cetera.

TODO: Maybe link to http://www.cs.ucdavis.edu/~rogaway/papers/relates.pdf for further reading

## 10.2 MD5

MD5 is a hash function designed by Ronald Rivest in 1991 as an extension of MD4. This hash function outputs 128-bit digests. Over the course of the years, the cryptographic community has repeatedly uncovered MD5's weaknesses. In 1993, Bert den Boer and Antoon Bosselaers published a paper demonstrating "pseudo-collisions" for the compression function of MD5. [dBB93] Dobbertin expanded upon this research and was able to produce collisions for the compression function. In 2004, based on Dobbertin's work, Xiaoyun Wang, Dengguo Feng, Xuejia Lai and Hongbo Yu showed that MD5 is vulnerable to real collision attacks. [LWdW05] The last straw came when Xiaoyun Wang et al. managed to generate colliding X.509 certificates and then presented a distinguishing attack on HMAC-MD5. [LWdW05] [WYW+09]

Nowadays, it is not recommended to use MD5 for generating digital signatures, but it is important to note that HMAC-MD5 is still a secure form of message authentication; however, it probably shouldn't be implemented in new cryptosystems.

Five steps are required to compute an MD5 message digest:

1. Add padding. First, 1 bit is appended to the message and then 0 bits are added to the end until the length is $448 \pmod{512}$.

2. Fill up the remaining 64 bits with the the length of the original message modulo $2^{64}$, so that the entire message is a multiple of 512 bits.

3. Initialize the state as four 32-bit words, A, B, C and D. These are initialized with constants defined in the spec.

4. Process the input in 512 bit blocks; for each block, run four “rounds” consisting of 16 similar operations each. The operations all consist of shifts, modular addition, and a specific nonlinear function, different for each round.

Once done, $A\|B\|C\|D$ is the output of the hash. This padding style combined with the concatenation at the end is what makes MD5 vulnerable to length extension attacks; more on that later.

In Python one can use the hashlib module to create an MD5 digest as follows:

```
import hashlib
hashlib.md5(b”crypto101”).hexdigest()
```

## 10.3 SHA-1

SHA-1 is another hash function from the MD4 family designed by the NSA, which produces a 160-bit digest. Just like MD5, SHA-1 is no longer considered secure for digital signatures. Many software companies and browsers, including Google Chrome, have started to retire support of the signature algorithm of SHA-1. On February 23, 2017 researchers from CWI Amsterdam and Google managed to produce a collision on the full SHA-1 function. [SBK+] In the past methods to cause collisions on reduced versions of SHA-1 have been published, including one by Xiaoyun Wang. “The SHAppening” demonstrated freestart collisions for SHA-1. A freestart collision allows one to pick the initial value known as the *initialization vector* at the start of the compression function. [SKP15]

Once again the hashlib Python module can be used to generate a SHA-1 hash:

```
import hashlib
hashlib.sha1(b”crypto101”).hexdigest()
```

## 10.4 SHA-2

SHA-2 is a family of hash functions including SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 and their digest sizes 224, 256, 384, 512, 224 and 256 respectively. These hash functions are based on the Merkle–Damgård construction and can be used for digital signatures, message authentication and random number generators. SHA-2 not only performs better than SHA-1, it also provides better security, because of its increase in collision resistance.

SHA-224 and SHA-256 were designed for 32-bit processor registers, while SHA-384 and SHA-512 for 64-bit registers. The 32-bit register variants will therefore run faster on a 32-bit CPU and the 64-bit variants will perform better on a 64-bit CPU. SHA-512/224 and SHA-512/256 are truncated versions of SHA-512 allowing use of 64-bit words with an output size equivalent to the 32-bit register variants (i.e., 224 and 256 digest sizes and better performance on a 64-bit CPU).

The following is a table that gives a good overview of the SHA-2 family:

| Hash function | Message size | Block size | Word size | Digest size |
|---|---|---|---|---|
| SHA-224 | $< 2^{64}$ | 512 | 32 | 224 |
| SHA-256 | $< 2^{64}$ | 512 | 32 | 256 |
| SHA-384 | $< 2^{128}$ | 1024 | 64 | 384 |
| SHA-512 | $< 2^{128}$ | 1024 | 64 | 512 |
| SHA-512/224 | $< 2^{128}$ | 1024 | 64 | 224 |
| SHA-512/256 | $< 2^{128}$ | 1024 | 64 | 256 |

You can hash an empty string with the hashlib module and compare digest sizes as follows:

```
>>> import hashlib
>>> len(hashlib.sha224(b””).hexdigest())
56
>>> len(hashlib.sha256(b””).hexdigest())
64
>>> len(hashlib.sha384(b””).hexdigest())
96
>>> len(hashlib.sha512(b””).hexdigest())
128
```

### Attacks on SHA-2

Several (pseudo-)collision and preimage attacks have been demonstrated using SHA-256 and SHA-512 with less rounds. It is important to note that by removing a certain amount of rounds one can't attack the entire algorithm. For instance, Somitra Kumar Sanadhya and Palash Sarkar were able to cause collisions with SHA-256 using 24 of 64 rounds (removing the last 40 rounds). [SS08]

## 10.5 Keccak and SHA-3

Keccak is a family of sponge functions designed by Guido Bertoni, Joan Daemen, Gilles Van Assche and Michaël Peeters, which won NIST's Secure Hash Algorithm Competition in 2012. Keccak has since been standardized in form of the SHA3-224, SHA3-256, SHA3-384 and SHA3-512 hash functions.

Although SHA-3 sounds like it might come from the same family as SHA-2, the two are designed very differently. SHA-3 is very efficient in hardware [Hua], but is relatively slow in software in comparison to SHA-2. [ECR] Later in the book, you will find the security aspects of SHA-3, such as preventing length extension attacks.

The SHA-3 hash functions were introduced in Python version 3.6 and can be used as follows:

```
import hashlib
hashlib.sha3_224(b”crypto101”).hexdigest()
hashlib.sha3_256(b”crypto101”).hexdigest()
hashlib.sha3_384(b”crypto101”).hexdigest()
hashlib.sha3_512(b”crypto101”).hexdigest()
```

## 10.6 Password storage

One of the most common use cases for cryptographic hash functions, and unfortunately one which is also completely and utterly broken, is password storage.

Suppose you have a service where people log in using a username and a password. You'd have to store the password somewhere, so that next time the user logs in, you can verify the password they supplied.

Storing the password directly has several issues. Besides an obvious timing attack in the string comparison, if the password database were to be compromised, an attacker would be able to just go ahead and read all of the passwords. Since many users re-use passwords, that's a catastrophic failure. Most user databases also contain their e-mail addresses, so it would be very easy to hi-jack a bunch of your user's accounts that are unrelated to this service.

### Hash functions to the rescue

An obvious approach would be to hash the password using a cryptographically secure hash function. Since the hash function is easy to compute, whenever the user provides their password, you can just compute the hash value of that, and compare that to what you stored in the database.

If an attacker were to steal the user database, they could only see the hash values, and not the actual passwords. Since the hash function is impossible for an attacker to inverse, they wouldn't be able to turn those back into the original passwords. Or so people thought.

## Rainbow tables

It turns out that this reasoning is flawed. The amount of passwords that people actually use is very limited. Even with very good password practices, they're strings somewhere between 10 and 20 characters, consisting mostly of things that you can type on common keyboards. In practice though, people use even worse passwords: things based on real words (`password`, `swordfish`), consisting of few symbols and few symbol types (`1234`), or with predictable modifications of the above (`passw0rd`).

To make matters worse, hash functions are the same everywhere. If a user re-uses the same password on two sites, and both of them hash the password using MD5, the values in the password database will be the same. It doesn't even have to be per-user: many passwords are extremely common (`password`), so many users will use the same one.

Keep in mind that a hash function is easy to evaluate. What if we simply try many of those passwords, creating huge tables mapping passwords to their hash values?

That's exactly what some people did, and the tables were just as effective as you'd expect them to be, completely breaking any vulnerable password store. Such tables are called *rainbow tables*. This is because they're essentially sorted lists of hash function outputs. Those outputs will be more or less randomly distributed. When written down in hexadecimal formats, this reminded some people of color specifications like the ones used in HTML, e.g. `#52f211`, which is lime green.

## Salts

The reason rainbow tables were so incredibly effective was because everyone was using one of a handful of hash functions. The same password would result in the same hash everywhere.

This problem was generally solved by using *salt*s. By mixing (appending or prepending[1]) the password with some random value before hashing it, you could produce completely different hash values out of the same hash function. It effectively turns a hash function into a whole family of related hash functions, with virtually identical security and performance properties, except with completely different output values.

The *salt* value is stored next to the password hash in the database. When the user authenticates using the password, you just combine the *salt* with the password, hash it, and compare it against the stored hash.

If you pick a sufficiently large (say, 160 bits/32 bytes), cryptographically random *salt*, you've completely defeated ahead-of-time attacks like rainbow tables. In order to successfully mount a rainbow table attack, an attacker would have to have a separate table for each of those *salt* values. Since even a single table was usually quite large, storing a large amount of them would be impossible. Even if an attacker would be able to store all that data, they'd still have to compute it first. Computing a single table takes a decent amount of time; computing $2^{160}$ different tables is impossible.

Many systems used a single *salt* for all users. While that prevented an ahead-of-time rainbow table attack, it still allowed attackers to attack all passwords simultaneously, once they knew the value of the *salt*. An attacker would simply compute a single rainbow table for that *salt*, and compare the results with the hashed passwords from the database. While this would have been prevented by using a different *salt* for each user, systems that use a cryptographic hash with a per-user *salt* are still considered fundamentally broken today; they are just *harder* to crack, but not at all secure.

[1] While you could also do this with XOR, it's needlessly more error-prone, and doesn't provide better results. Unless you zero-pad both the password and the *salt*, you might be truncating either one.

Perhaps the biggest problem with *salt*s is that many programmers were suddenly convinced they were doing the right thing. They'd heard of broken password storage schemes, and they knew what to do instead, so they ignored all talk about how a password database could be compromised. They weren't the ones storing passwords in plaintext, or forgetting to *salt* their hashes, or re-using *salt*s for different users. It was all of those other people that didn't know what they were doing that had those problems. Unfortunately, that's not true. Perhaps that's why broken password storage schemes are still the norm.

## Modern attacks on weak password systems

To a modern attack, *salt*s quite simply don't help. Modern attacks take advantage of the fact that the hash function being used is easy to compute. Using faster hardware, in particular video cards, we can simply enumerate all of the passwords, regardless of *salt*.

TODO: more concrete performance numbers about GPUs

*Salt*s may make precomputed attacks impossible, but they do very little against an attacker that actually knows the *salt*. One approach you might be inclined to take is to attempt to hide the *salt* from the attacker. This typically isn't very useful: if an attacker can manage to access the database, attempts to hide the *salt* are unlikely to be successful. Like many ineffective home-grown crypto schemes, this only protects against an incredibly improbable event. It would be much more useful to just use a good password store to begin with, than trying to fix a broken one.

## So where do we go from here?

In order to protect passwords, you need a (low-entropy) *key derivation function* (page 137). We'll discuss them in more detail in a future chapter.

While key derivation functions can be built using cryptographic hash functions, they have very different performance properties. This is a common pattern: while cryptographic hash functions are incredibly important primitives for building secure tools (such as key derivation functions or message authentication algorithms), they are routinely abused *as* those tools themselves. In the rest of this chapter, we will see other examples of how cryptographic hash functions can be used and abused.

## 10.7 Length extension attacks

In many hash functions, particularly the previous generations, the internal state kept by the hash function is used as the digest value. In some poorly engineered systems, that causes a critical flaw: if an attacker knows $H(M_1)$, it's very simple to compute $H(M_1 \| M_2)$, without actually knowing the value of $M_1$. Since you know $H(M_1)$, you know the state of the hash function after it's hashed $M_1$. You can use that to reconstruct the hash function, and ask it to hash more bytes. Setting the hash function's internal state to a known state you got from somewhere else (such as $H(M_1)$) is called *fixation*.

For most real-world hash functions, it's a little bit more complicated than that. They commonly have a padding step that an attacker needs to recreate. MD5 and SHA-1 have the same padding step. It's fairly simple, so we'll go through it:

1. Add a 1 bit to the message.
2. Add zero bits until the length is $448 \pmod{512}$.
3. Take the total length of the message, before padding, and add it as a 64-bit integer.

For the attacker to be able to compute $H(M_1 \| M_2)$ given $H(M_1)$, the attacker needs to fake that padding, as well. The attacker will actually compute $H(M_1 \| G \| M_2)$, where $G$ is the *glue padding*, called that way because it *glues* the two messages together. The hard part is knowing the length of the message $M_1$.

In many systems, the attacker can actually make fairly educated guesses about the length of $M_1$, though. As an example, consider the common (broken) example of a secret-prefix authentication code. People send messages $M_i$, authenticated using $A_i = H(S\|M_i)$, where $S$ is a shared secret. We'll see (and break) this MAC algorithm in a future section.

It's very easy for the recipient to compute the same function, and verify the code is correct. Any change to the message $M_i$ will change the value of $A_i$ drastically, thanks to the avalanche effect. Unfortunately, it's quite easy for attackers to forge messages. Since the MAC is usually sent together with the original message, the attacker knows the length of the original message. Then, the attacker only has to guess at the length of the secret, which is often fixed as part of the protocol, and, even if it isn't, the attacker will probably get in a hundred tries or less. Contrast this with guessing the secret itself, which is impossible for any reasonably chosen secret.

There are secure authentication codes that can be designed using cryptographic hash functions: this one just isn't it. We'll see better ones in a later chapter.

Some hash functions, particularly newer ones such as SHA-3 competition finalists, do not exhibit this property. The digest is computed from the internal state, instead of using the internal state directly.

This makes the SHA-3-era hash functions not only a bit more fool-proof, but also enables them to produce simpler schemes for message authentication. (We'll elaborate on those in a later chapter.) While length extension attacks only affected systems where cryptographic hash functions were being abused in the first place, there's something to be said for preventing them anyway. People will end up making mistakes, we might as well mitigate where we can.

TODO: say why this prevents meet in the middle attacks?

## 10.8 Hash trees

Hash trees are trees[2] where each node is identified by a hash value, consisting of its contents and the hash value of its ancestor. The root node, not having an ancestor, simply hashes its own contents.

This definition is very wide: practical hash trees are often more restricted. They might be binary trees[3], or perhaps only leaf nodes carry data of their own, and parent nodes only carry derivative data. Particularly these restricted kinds are often called Merkle trees.

Systems like these or their variants are used by many systems, particularly distributed systems. Examples include distributed version control systems such as Git, digital currencies such as Bitcoin, distributed peer-to-peer networks like Bittorrent, and distributed databases such as Cassandra.

## 10.9 Remaining issues

We've already illustrated that hash functions, by themselves, can't authenticate messages, because anyone can compute them. Also, we've illustrated that hash functions can't be used to secure passwords. We'll tackle both of these problems in the following chapters.

While this chapter has focused heavily on what hash functions *can't* do, it can't be stressed enough that they are still incredibly important cryptographic primitives. They just happen to be commonly *abused* cryptographic primitives.

[2] Directed graphs, where each node except the root has exactly one ancestor.

[3] Each non-leaf node has no more than two children

11

# Message authentication codes

## 11.1 Description

A MAC is a small bit of information that can be used to check the authenticity and the integrity of a message. These codes are often called "tags". A MAC algorithm takes a message of arbitrary length and a secret key of fixed length, and produces the tag. The MAC algorithm also comes with a verification algorithm that takes a message, the key and a tag, and tells you if the tag was valid or not. (It is not always sufficient to just recompute a tag and check if they are the same; many secure MAC algorithms are randomized, and will produce different tags every time you apply them.)

Note that we say "message" here instead of "plaintext" or "ciphertext". This ambiguity is intentional. In this book we're mostly interested in MACs as a way to achieve authenticated encryption, so the message will always be a ciphertext. That said, there's nothing wrong with a MAC being applied to a plaintext message. In fact, we will be seeing examples of secure authenticated encryption schemes that explicitly allow for authenticated (but not encrypted) information to be sent along with the authenticated ciphertext.

Often, when you just want to talk about the authenticity and integrity of a particular message, it may be more practical to use a *signature algorithm*, which we'll talk about in a later chapter. For now, all you need to know is that the term "signature" is normally reserved for asymmetric algorithms, whereas this chapter deals with symmetric algorithms.

## Secure MACs

We haven't quite defined yet exactly which properties we want from a secure MAC.

We will be defending against an active attacker. The attacker will be performing a *chosen message attack*. That means that an attacker will ask us the tag for any number of messages $m_i$, and we'll answer truthfully with the appropriate tag $t_i$.

An attacker will then attempt to produce an *existential forgery*, a fancy way of saying that they will produce some new valid combination of $(m, t)$. The obvious target for the attacker is the ability to produce valid tags $t'$ for new messages $m'$ of their choosing. We will also consider the MAC insecure if an attacker can compute a new, different valid tag $t'$ for a message $m_i$ that we previously gave them a valid tag for.

## Why does a MAC take a secret key?

If you've had to deal with verifying the integrity of a message before, you may have used checksums (like CRC32 or Adler32) or even cryptographic hashes (like the SHA family) in order to compute a checksum for the message (depending on the algorithm and who you're talking to, they may have called it "hash" or "digest", too).

Let's say that you're distributing a software package. You have some tarballs with source code in them, and maybe some binary packages for popular operating systems. Then you put some (cryptographically secure!) hashes right next to them, so that anyone who downloads them can verify the hashes and be confident that they downloaded what they think they downloaded.

Of course, this scheme is actually totally broken. Computing those hashes is something everyone can do. You're even relying on that fact for your user to be able to verify their download. That also means that an attacker that modified any of the downloads can just compute the hash again for the modified download and save that value. A user downloading the modified file will compute its hash and compare it against the modified hash, and conclude that the download worked. The scheme provided no help whatsoever against an attacker modifying the download, either as stored, or in transit.

In order to do this securely, you would either apply a signature algorithm to the binaries directly, or by signing the digests, as long as the hash function used to produce the digest is secure against second-preimage attacks. The important difference is that producing a signature (using either a pre-shared key with your users, or, preferably, a public-key signature algorithm) is *not* something that an attacker can do. Only someone who has the secret keys can do that.

## 11.2 Combining MAC and message

As we've mentioned before, unauthenticated encryption is bad. That's why we introduced MACs. Of course, for a MAC to be useful, it has to make it to the recipient. Since we're explicitly talking about authenticating encryption, now, we'll stop using the word "message" and instead use the less ambiguous "plaintext" and "ciphertext".

There are three common ways to combine a ciphertext with a MAC.

1. Authenticate and encrypt. You authenticate and en-

crypt the plaintext separately. This is how SSH does it. In symbols: $C = E(K_C, P)$, $t = MAC(K_M, P)$, and you send both ciphertext $C$ and tag $t$.

2. Authenticate, then encrypt. You authenticate the plaintext and then encrypt the combination of the plaintext and the authentication tag. This is how TLS usually does it. In symbols: $t = MAC(K_M, P)$, $C = E(K_C, P\|t)$, and you only send $C$. (You don't need to send $t$, because it's already an encrypted part of $C$.)

3. Encrypt, then authenticate. You encrypt the plaintext, compute the MAC of that ciphertext. This is how IPSec does it. In symbols: $C = E(K_C, P)$, $t = MAC(K_M, C)$, and you send both $C$ and $t$.

All of these options were studied and compared extensively. [Kra01] [BN07] We now know that out of all of these, encrypt-then-authenticate is unequivocally the best option. It's so emphatically the best option that Moxie Marlinspike, a well-respected information security researcher, has a principle called "The Cryptographic Doom Principle" for any system that does *not* follow this pattern [Mar11]. Moxie claims that any system that does anything before checking the MAC is doomed. Both authenticate-and-encrypt and authenticate-then-encrypt require you to decrypt something before you can verify the authentication.

## Authenticate-then-encrypt

Authenticate-then-encrypt is a poor choice, but it's a subtle poor choice. It can still be provably secure, but only under certain conditions. [Kra01]

At first sight, this scheme appears to work. Sure, you have to decrypt before you can do anything, but to many cryptographers, including the designers of TLS, this did not appear to pose a problem.

In fact, prior to rigorous comparative study of different composition mechanisms, many preferred this setup.

In a critique of IPSec, Schneier and Ferguson, two veteran cryptographers, considered IPSec's use of encrypt-then-authenticate was a flaw, preferring TLS's authenticate-then-encrypt. [FS99] While they may have had a plausible (albeit mostly heuristic) argument for the time, this criticism is completely superseded by the *provable* security of encrypt-then-authenticate schemes. [Kra01] [BN07]

TODO: Explain Vaudenay CBC attack [Vau]

### Authenticate-and-encrypt

Authenticate-and-encrypt has some serious problems. Since the tag authenticates the plaintext and that tag is part of the transmitted message, an attacker will be able to recognize two plaintext messages are the same because their tags will also be the same. This essentially leads to the same problem we saw with ECB mode, where an attacker can identify identical blocks. That's a serious problem, even if they can't decrypt those blocks.

TODO: Explain how this works in SSH (see Moxie's Doom article)

## 11.3 A naive attempt with hash functions

Many ways of constructing MACs involve hash functions. Perhaps one of the simplest ways you could imagine doing that is to just prefix the message with the secret key and hash the whole thing:

$$t = H(k \| m)$$

This scheme is most commonly called "Prefix-MAC", because it is a MAC algorithm that works by using the secret key as a prefix.

The cryptographically secure hash function $H$ guarantees a few things that are important to us here:

- The tag $t$ will be easy to compute; the hash function $H$ itself is typically very fast. In many cases we can compute the common key part ahead of time, so we only have to hash the message itself.
- Given any number of tags, there is no way for an attacker to "invert" the hash function to recover $k$, which would allow them to forge arbitrary messages.
- Given any number of tags, there is no way for an attacker to "rewind" the hash function to recover $H(k)$, which may allow them to forge *almost* arbitrary messages.

One small caveat: we're assuming that the secret key $k$ has enough entropy. Otherwise, we have the same issue that we had for password storage using hash functions: an attacker could just try every single $k$ until one of them matches. Once they've done that, they've almost certainly found the correct $k$. That's not really a failure of the MAC though: if your secret key contains so little entropy that it's feasible for an attacker to try all of them, you've already lost, no matter which MAC algorithm you pick.

## Breaking prefix-MAC

Despite being quite common, this MAC is actually completely insecure for most (cryptographically secure!) hash functions $H$, including SHA-2.

As we saw in the chapter on hash functions, many hash functions, such as MD5, SHA-0, SHA-1 and SHA-2, pad the message with a predictable padding before producing the output digest. The output digest is the same thing as the internal state of the hash function. That's a problem: the attacker can use those properties to forge messages.

First, they use the digest as the internal state of the hash function. That state matches the state you get when you hash $k \| m \| p$, where $k$ is the secret key, $m$ is the message, and $p$ is that predictable padding. Now, the attacker gets the hash function to consume some new bytes: the attacker's chosen message $m'$. The internal state of the hash function is now what you get when you feed it $k\|m\|p\|m'$. Then, the attacker tells the hash function to produce a digest. Again, the hash function appends a padding, so we're now at $k\|m\|p\|m'\|p'$. The attacker outputs that digest as the tag. That is *exactly* the same thing as what happens when you try to compute the tag for the message $m\|p\|m'$ under the secret key $k$. So, the attacker has successfully forged a tag for a new message, and, by our definition, the MAC is insecure.

This attack is called a length extension attack, because you are extending a valid message. The padding in the middle $p$, which started out as the padding for the original message but has become just some data in the middle, is called *glue padding*, because it glues the original message $m$ and the attacker's message $m'$ together.

This attack might sound a little academic, and far from a practical problem. We may have proven that the MAC is insecure by our definition, but the only tags the attacker can successfully forge are for very limited modifications of real messages. Specifically, the attacker can only forge tags for a message that consists of a message we sent, followed by some binary junk, followed by something the attacker chooses. However, it turns out that for many systems, this is plenty to result in real breaks. Consider the following Python code that parses a sequence of key-value pairs that look like `k1=v1&k2=v2&...`:[1]

```
def parse(s):
    pairs = s.split(”&”)
    parsed = {}
    for pair in pairs:
        key, value = pair.split(”=”)
```

(continues on next page)

[1] I realize there are briefer ways to write that function. I am trying to make it comprehensible to most programmers; not pleasing to advanced Pythonistas.

(continued from previous page)

```
        parsed[key] = value
    return parsed
```

The parsing function only remembers the last value for a given key: previous values in the dictionary are overwritten. As a result, an attacker mounting a length extension attack can effectively control the parsed dictionary entirely.

If you're thinking that this code has many issues; sure, it does. For example, it doesn't handle escaping correctly. But even if it did, that wouldn't really fix the length extension attack problem. Most parsing functions will perfectly happily live with that binary junk in the middle. Hopefully it convinces you that there is in fact a pretty good chance that an attacker can produce messages with valid tags that say something entirely different from what you intended.

The prefix-MAC construction is actually secure with many current (SHA-3-era) hash functions, such as Keccak and BLAKE(2). The specifications for these hash functions even recommend it as a secure and fast MAC. They use various techniques to foil length extension attacks: for example, BLAKE keeps track of the number of bits that have been hashed so far, while BLAKE2 has a finalization flag that marks a specific block as the last.

## Variants

Issues with prefix-MAC has tempted people to come up with all sorts of clever variations. For example, why not add the key to the end instead of the beginning ($t = H(m\|k)$, or "suffix-MAC", if you will)? Or maybe we should append the key to both ends for good measure ($t = H(k\|m\|k)$, "sandwich-MAC" perhaps?)?

For what it's worth, both of these are at least better than prefix-MAC, but both of these have serious issues. For example, a suffix-MAC system is more vulnerable to weaknesses in the underlying hash function; a successful collision attack breaks the MAC. Sandwich-MAC has other, more complex issues.

Cryptography has produced much stronger MACs, which we'll see in the next few sections. There are no good reasons not to use them.

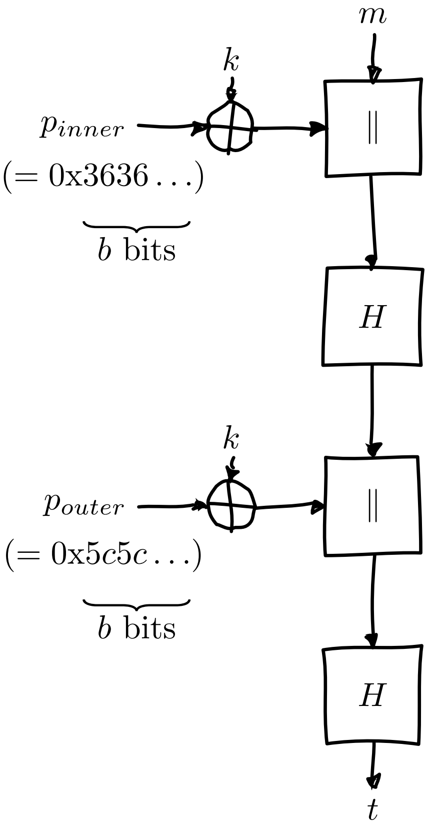
## 11.4 HMAC

HMAC is a standard to produce a MAC with a cryptographic hash function as a parameter. It was introduced in 1996 in a paper by Bellare, Canetti and Krawczyk. Many protocols at the time implemented their own attempt at message authentication using hash functions. Most of these attempts failed. The goal of that paper specifically was to produce a provably secure MAC that didn't require anything beyond a secret key and a hash function.

One of the nice features of HMAC is that it has a fairly strong security proof. As long as the underlying hash function is a pseudorandom function, HMAC itself is also a pseudorandom function. The underlying hash function doesn't even have to be collision resistant for HMAC to be a secure MAC. [Bel06] This proof was introduced after HMAC itself, and matched real-world observations: even though MD5 and to a lesser extent SHA-0 had serious collision attacks, HMAC constructions built from those hash functions still appeared to be entirely secure.

The biggest difference between HMAC and prefix-MAC or its variants is that the message passes through a hash function twice, and is combined with the key before each pass. Visually, HMAC looks like this:

The only surprising thing here perhaps are the two constants $p_{inner}$ (the inner padding, one hash function's block length worth of `0x36` bytes) and $p_{outer}$ (the outer padding, one block length worth of `0x5c` bytes). These are necessary for the security proof of HMAC to work; their particular values aren't very important, as long as the two constants are different.


$m$
$k$
$p_{inner}$
$(= 0x3636\ldots)$
$\underbrace{\phantom{xxxx}}_{b \text{ bits}}$
$\|$
$H$
$k$
$p_{outer}$
$(= 0x5c5c\ldots)$
$\underbrace{\phantom{xxxx}}_{b \text{ bits}}$
$\|$
$H$
$t$

The two pads are XORed with the key before use. The result is either prepended to the original message (for the inner padding $p_{inner}$) or to the intermediate hash output (for the outer padding $p_{outer}$). Because they're prepended, the internal state of the hash function after processing the prefixes can be computed ahead of time, shaving a few cycles off the MAC computation time.

## 11.5 One-time MACs

So far, we've always assumed that MAC functions can be used with a single key to produce secure MACs for a very large number of messages. By contrast, *one-time MAC*s are

MAC functions that can only securely be used once with a single key. That might sound like a silly idea, since we've already talked about regular secure MACs. An algorithm that only works once just seems objectively worse. However, they have several big advantages:

- They can be incredibly fast to evaluate, even for very large messages.
- They have a compelling security proof based on the information content of the tag.
- A construction exists to turn a *one-time MAC* into a secure multiple-use MAC, removing the principal problem.

A typical simple example of such *one-time MAC*s consists of a simple multiplication and addition modulo some large prime $p$. In this case, the secret key consists of two truly random numbers $a$ and $b$, both between 1 and $p$.

$$t \equiv m \cdot a + b \pmod p$$

This simple example only works for one-block messages $m$, and some prime $p$ slightly bigger than the biggest $m$. It can be extended to support bigger messages $M$ consisting of blocks $m_i$ by using a message-specific polynomial $P$:

$$t \equiv \underbrace{(m_n \cdot a^n + \cdots + m_1 \cdot a)}_{P(M,a)} + b \pmod p$$

This might look like a lot of computation, but this polynomial can be efficiently evaluated by iteratively factoring out the common factor $a$ (also known as Horner's rule):

$$P(M, a) \equiv a \cdot (a \cdot (a \cdot (\cdots) + m_2) + m_1) + b \pmod p$$

By computing each multiplication modulo $p$, the numbers will remain conveniently small.

In many ways, a *one-time MAC* is to authentication what a one-time pad is to encryption. The security argument is similar: as long as the key is only used once, an attacker learns no information about the key or the message, because they are being irreversibly mixed. This demonstrates that the MAC is secure against attackers trying to produce existential forgeries, even when that attacker has infinite computational power.

Also like a one-time pad, the security argument relies on two very important properties about the keys $a, b$:

- They have to be truly random.
- They have to be used at most once.

## Re-using $a$ and $b$

We'll illustrate that our example MAC is insecure if it is used to authenticate two messages $m_1, m_2$ with the same key $(a, b)$:

$$\begin{aligned} t_1 &\equiv m_1 \cdot a + b \pmod{p} \\ t_2 &\equiv m_2 \cdot a + b \pmod{p} \end{aligned}$$

An attacker can reconstruct $a, b$ with some simple modular arithmetic:[2]

$$\begin{aligned} t_1 - t_2 &\equiv (m_1 \cdot a + b) - (m_2 \cdot a + b) \pmod{p} \\ &\Downarrow \text{(remove parentheses)} \\ t_1 - t_2 &\equiv m_1 \cdot a + b - m_2 \cdot a - b \pmod{p} \\ &\Downarrow (b \text{ and } -b \text{ cancel out}) \\ t_1 - t_2 &\equiv m_1 \cdot a - m_2 \cdot a \pmod{p} \\ &\Downarrow (\text{factor out } a) \\ t_1 - t_2 &\equiv a \cdot (m_1 - m_2) \pmod{p} \\ &\Downarrow (\text{flip sides, multiply by inverse of } (m_1 - m_2)) \\ a &\equiv (t_1 - t_2)(m_1 - m_2)^{-1} \pmod{p} \end{aligned}$$

[2] For a refresher on modular arithmetic, including an explanation of the modular inverse, please refer to *the appendix* (page 186).

Plugging $a$ into either the equation for $t_1$ or $t_2$ gets $b$:

$$\begin{gathered} t_1 \equiv m_1 \cdot a + b \pmod p \\ \Downarrow \text{(reorder terms)} \\ b \equiv t_1 - m_1 \cdot a \pmod p \end{gathered}$$

As you can see, as with one-time pads, re-using the key even once leads to a complete failure of the cryptosystem to preserve privacy or integrity, as the case may be. As a result, *one-time MAC*s are a bit dangerous to use directly. Fortunately, this weakness can be solved with a construction called a *Carter-Wegman MAC*, which we'll see in the next section.

## 11.6 Carter-Wegman MAC

As we've already stated, the obvious problem with *one-time MAC*s is their limited practicality. Fortunately, it turns out that there is a construction, called a *Carter-Wegman MAC*, that turns any secure one-time MAC into a secure many-time MAC while preserving most of the performance benefit.

The idea behind a *Carter-Wegman MAC* is that you can use a *one-time MAC* $O$ to produce a tag for the bulk of the data, and then encrypt a *nonce* $n$ with a pseudorandom function $F$, such as a block cipher, to protect that one-time tag:

$$CW((k_1, k_2), n, M) = F(k_1, n) \oplus O(k_2, M)$$

As long as $F$ is a secure pseudorandom function, the *nonce*'s encryption is totally unpredictable. In the eyes of an attacker, that means the XOR operation will randomly flip the bits of the *one-time MAC* tag $O(k_2, M)$. Because this masks the real value of the *one-time MAC* tag, the attacker can not perform the algebraic tricks we saw for *one-time MAC*s recovering the key when it is used more than once.

Keep in mind that while *Carter-Wegman MAC*s take two distinct keys $k_1$ and $k_2$, and that *Carter-Wegman MAC*s are related to *one-time MAC*s, some of which also take two distinct keys $a$ and $b$, they are not the same two keys. The Carter-Wegman MAC's $k_2$ is the only key passed to the fast *one-time MAC* $O$. If that fast *one-time MAC* is our earlier example that takes two keys $a$ and $b$, that $k_2$ would have to get split up into those two keys. The *Carter-Wegman MAC* key would then be $(k_1, k_2) = (k_1, (a, b))$.

You can tell how a *Carter-Wegman MAC* exploits the benefits of both kinds of MACs by considering the two terms of the equation separately. In $F(k_1, n)$, $F$ is just a regular pseudorandom function, such as a block cipher. It is quite slow by comparison to the one-time MAC. However, its input, the *nonce*, is very small. The unpredictable output of the block cipher masks the output of the *one-time MAC*. In the second term, $O(k_2, M)$, the large input message $M$ is only handled by the very fast *one-time MAC* $O$.

These constructions, in particular Poly1305-AES, currently represent some of the state of the art in MAC functions. The paper ([BHK+99]) and RFC ([BHK+]) for an older, related MAC function called UMAC may also be good sources of extra background information, since they go into extensive details of the hows and whys of a practical *Carter-Wegman MAC*.

## 11.7 Authenticated encryption modes

So far, we've always clearly distinguished encryption from authentication, and explained the need for both. The majority of secure connections that are set up every day have that distinction as well: they treat encryption and authentication as fundamentally different steps.

Alternatively, we could make authentication a fundamental part of the *mode of operation*. After all, we've already seen that unauthenticated encryption is virtually never what you want; it is, at best, something you occasionally have to live with. It makes sense to use constructions that not only guarantee the privacy of an arbitrary stream, but also its integrity.

As we've already seen, many of the methods of composing authentication and encryption are inherently insecure. By doing that in a fixed, secure way such as a properly designed authenticated encryption mode, an application developer no longer has to make that choice, which means they also can't inadvertently make the *wrong* choice.

## AEAD

AEAD is a feature of certain modes of authenticated encryption. Such modes of operation are called *AEAD mode*s. It starts with the premise that many messages actually consist of two parts:

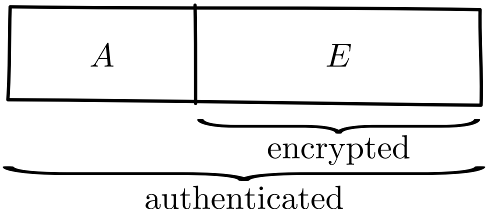
- The actual content itself
- Metadata: data *about* the content

In many cases the metadata should be plaintext, but the content itself should be encrypted. The entire message should be authenticated: it should not be possible for an attacker to mess with the metadata and have the resulting message still be considered valid.

Consider an e-mail alternative as an example cryptosystem. The metadata about the content might contain the intended recipient. We definitely want to encrypt and authenticate the content itself, so that only the recipient can read it. The metadata, however, has to be in plaintext: the e-mail servers performing the message delivery have to know which recipient to send the message to.

Many systems would leave this metadata unauthenticated, allowing attackers to modify it. In our case, that looks like it may just lead to messages being delivered to the wrong inbox. That also means that an attacker can force e-mail to be delivered to the wrong person, or not delivered at all.

*AEAD mode*s address this issue by providing a specified way to add metadata to encrypted content, so that the whole of the encrypted content and the metadata is authenticated, and not the two pieces separately:



## 11.8 OCB mode

This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

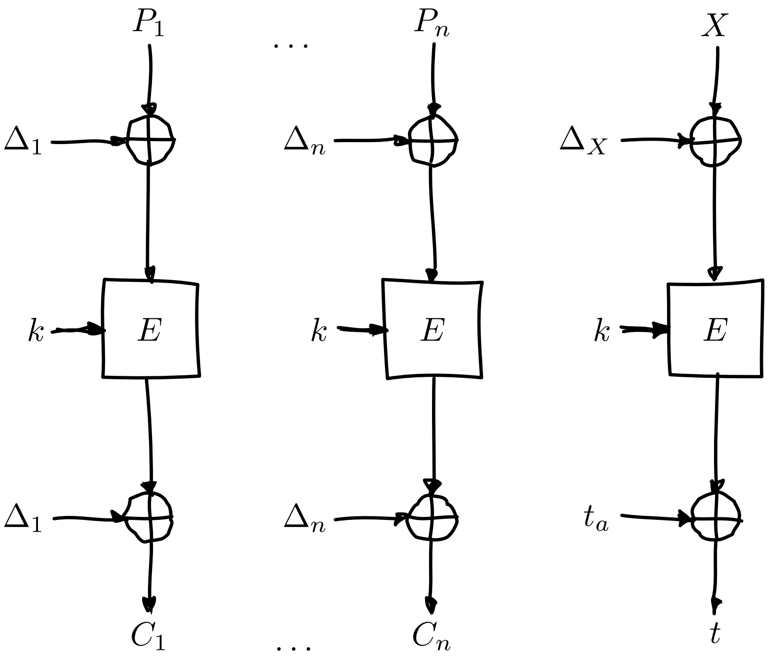
Usually, you will want to use a much more high level cryptosystem, such as OpenPGP, NaCl or TLS.

*OCB mode* is an *AEAD mode* of operation. It is one of the earliest developed *AEAD mode*s.

As you can see, most of this scheme looks quite similar to *ECB mode*. The name OCB is quite similar to electronic codebook, as well. OCB does not share the security issues ECB mode has, however, as there are several important differences, such as the offsets $\Delta_i$ introduced in each individual block encryption.

Being an *AEAD mode*, *OCB mode* provides a cryptographically secure authentication tag $t$, which is built from $X$, a very simple (not cryptographically secure by itself) checksum of the plaintext. There is also another, separate tag $t_a$, which authenticates the AEAD associated data. That associated data tag $t_a$ is computed as follows:

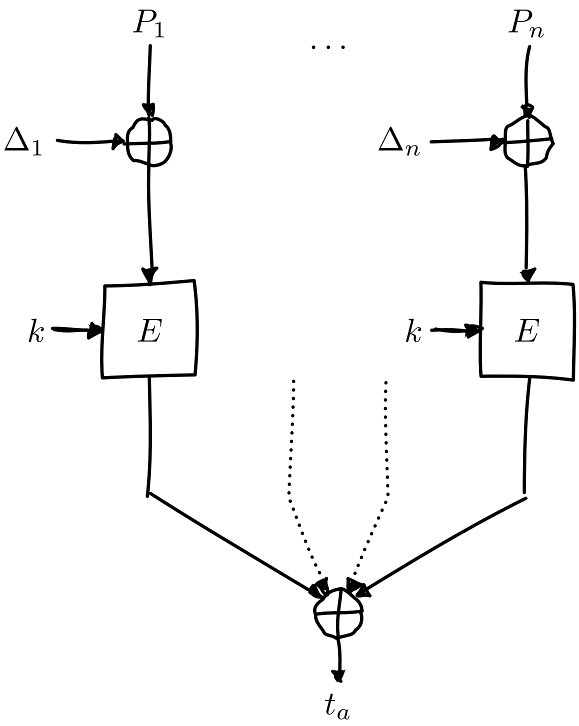
This design has a number of interesting properties. For example, it is very fast: only requiring roughly one block cipher operation per encrypted or associate data block, as well as one additional block cipher operation for the final tag. The offsets ($\Delta_i$) are also extremely easy to compute. The



checksum block $X$ is just all of the plaintext blocks $P_i$ XORed together. Finally, *OCB mode* is easy to compute in parallel; only the final authentication tag is dependent on all the preceding information.

*OCB mode* also comes with a built-in padding scheme: it behaves slightly differently when the plaintexts or authentication text is not exactly a multiple of the block size. This means that, unlike with PKCS#5/PKCS#7 padding, there isn't an entire block of "wasted" padding if the plaintext happens to be a multiple of the block size.

Despite having several interesting properties going for it, *OCB mode* has not received as much attention as some of the alternatives; one of the main reasons being that it is patent encumbered. Even though a number of patent licenses are available, including a free-of-charge one for open source software, this does not appear to have significantly impacted how much *OCB mode* is used in the field. [Rog]



## 11.9 GCM mode

This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

Usually, you will want to use a much more high level cryptosystem, such as OpenPGP, NaCl or TLS.

*GCM mode* is an *AEAD mode* with an unfortunate case of RAS (redundant acronym syndrome) syndrome: GCM itself stands for "Galois Counter Mode". It is formalized in a NIST Special Publication [gcm07] and roughly boils down to a combination of classical CTR mode with a *Carter-Wegman*

*MAC*. That MAC can be used by itself as well, which is called *GMAC*.

### Authentication

*GCM mode* (and by extension *GMAC*)

12

# Signature algorithms

## 12.1 Description

A signature algorithm is the public-key equivalent of a message authentication code. It consists of three parts:

1. a key generation algorithm, which can be shared with other *public-key algorithm*s
2. a signature generation algorithm
3. a signature verification algorithm

Signature algorithms can be built using encryption algorithms. Using the private key, we produce a value based on the message, usually using a cryptographic hash function. Anyone can then use the public key to retrieve that value, compute what the value should be from the message, and compare the two to verify. The obvious difference between this and *public-key encryption* is that in signing, the private key is used to produce the message (in this case the signature) and the public key is used to interpret it, which is the opposite of how encryption and decryption work.

The above explanation glosses over many important details. We'll discuss real schemes in more detail below.

## 12.2 RSA-based signatures

### PKCS#1 v1.5

TODO (see #48)

### PSS

TODO (see #49)

## 12.3 DSA

The Digital Signature Algorithm (DSA) is a US Federal Government standard for digital signatures. It was first proposed by the National Institute of Standards and Technology (NIST) in 1991, to be used in the Digital Signature Standard (DSS). The algorithm is attributed to David W. Kravitz, a former technical advisor at the NSA.

DSA key generation happens in two steps. The first step is a choice of parameters, which can be shared between users. The second step is the generation of public and private keys for a single user.

### Parameter generation

We start by picking an approved cryptographic hash function $H$. We also pick a key length $L$ and a prime length $N$. While the original DSS specified that $L$ be between 512 and 1024, NIST now recommends a length of 3072 for keys with a security lifetime beyond 2030. As $L$ increases, so should $N$.

Next we choose a prime $q$ of length $N$ bits; $N$ must be less than or equal to the length of the hash output. We also pick an L-bit prime $p$ such that $p - 1$ is a multiple of $q$.

The last part is the most confusing. We have to find a number $g$ whose *multiplicative order* (page 201) $\pmod p$ is $q$. The easy way to do this is to set $g \equiv 2^{(p-1)/q} \pmod p$. We can try another number greater than 2, and less than $p-1$, if $g$ comes out to equal 1.

Once we have parameters $(p, q, g)$, they can be shared between users.

## Key generation

Armed with parameters, it's time to compute public and private keys for an individual user. First, select a random $x$ with $0 < x < q$. Next, calculate $y$ where $y \equiv g^x \pmod p$. This delivers a public key $(p, q, g, y)$, and private key $x$.

## Signing a message

In order to sign a message, the signer picks a random $k$ between 0 and $q$. Picking that $k$ turns out to be a fairly sensitive and involved process; but we'll go into more detail on that later. With $k$ chosen, they then compute the two parts of the signature $r, s$ of the message $m$:

$$r \equiv (g^k \pmod p) \pmod q$$

$$s \equiv k^{-1}(H(m) + xr) \pmod q$$

If either of these happen to be 0 (a rare event, with 1 in $q$ odds, and $q$ being a pretty large number), pick a different $k$.

TODO: Talk about k$^{-1}$, the modular inverse (see #52)

## Verifying a signature

Verifying the signature is a lot more complex. Given the message $m$ and signature $(r, s)$:

$$w \equiv s^{-1} \pmod q$$

$$u_1 \equiv wH(m) \pmod q$$

$$u_2 \equiv wr \pmod{q}$$

$$v \equiv (g^{u_1} y^{u_2} \pmod{p}) \pmod{q}$$

If the signature is valid that final result $v$ will be equal to $r$, the second part of the signature.

## The trouble with $k$

While there is nothing wrong with DSA done right, it's very easy to get it wrong. Furthermore, DSA is quite sensitive: even a small implementation mistake results in a broken scheme.

In particular, the choice of the signature parameter $k$ is critical. The requirements for this number are among the strictest of all random numbers in cryptographic algorithms. For example, many algorithms require a *nonce*. A *nonce* just has to be unique: you can use it once, and then you can never use it again. It doesn't have to be secret. It doesn't even have to be unpredictable. A *nonce* can be implemented by a simple counter, or a monotonic clock. Many other algorithms, such as *CBC mode*, use an initialization vector. It doesn't have to be unique: it only has to be unpredictable. It also doesn't have to be secret: initialization vectors are typically tacked on to the ciphertext. DSA's requirements for the $k$ value are a combination of all of these:

- It has to be unique.
- It has to be unpredictable.
- It has to be secret.

Muddle with any of these properties, and an attacker can probably retrieve your secret key, even with a modest amount of signatures. For example, an attacker can recover the secret key knowing only a few bits of $k$, plus a large amount of valid signatures. [NS00]

It turns out that many implementations of DSA don't even get the uniqueness part right, happily reusing $k$ values. That allows a direct recovery of the secret key using basic arithmetic. Since this attack is much simpler to understand, very commonly applicable, and equally devastating, we'll discuss it in detail.

Suppose that an attacker sees multiple signatures $(r_i, s_i)$, for different messages $m_i$, all with the same $k$. The attacker picks any two signatures $(r_1, s_1)$ and $(r_2, s_2)$ of messages $m_1$ and $m_2$ respectively. Writing down the equations for $s_1$ and $s_2$:

$$s_1 \equiv k^{-1}(H(m_1) + xr_1) \pmod{q}$$

$$s_2 \equiv k^{-1}(H(m_2) + xr_2) \pmod{q}$$

The attacker can simplify this further: $r_1$ and $r_2$ must be equal, following the definition:

$$r_i \equiv g^k \pmod{q}$$

Since the signer is reusing $k$, and the value of $r$ only depends on $k$, all $r_i$ will be equal. Since the signer is using the same key, $x$ is equal in the two equations as well.

Subtract the two $s_i$ equations from each other, followed by some other arithmetic manipulations:

$$\begin{aligned}
s_1 - s_2 \equiv & \quad k^{-1}(H(m_1) + xr) - k^{-1}(H(m_2) + xr) \pmod{q} \\
\equiv & \quad k^{-1}\left((H(m_1) + xr) - (H(m_2) + xr)\right) \pmod{q} \\
\equiv & \quad k^{-1}(H(m_1) + xr - H(m_2) - xr) \pmod{q} \\
\equiv & \quad k^{-1}(H(m_1) - H(m_2)) \pmod{q}
\end{aligned}$$

This gives us the simple, direct solution for $k$:

$$k \equiv (H(m_1) - H(m_2))\,(s_1 - s_2)^{-1} \pmod{q}$$

The hash values $H(m_1)$ and $H(m_2)$ are easy to compute. They're not secret: the messages being signed are public.

The two values $s_1$ and $s_2$ are part of the signatures the attacker saw. So, the attacker can compute $k$. That doesn't give him the private key $x$ yet, though, or the ability to forge signatures.

Let's write the equation for $s$ down again, but this time thinking of $k$ as something we know, and $x$ as the variable we're trying to solve for:

$$s \equiv k^{-1}(H(m) + xr) \pmod{q}$$

All $(r, s)$ that are valid signatures satisfy this equation, so we can just take any signature we saw. Solve for $x$ with some algebra:

$$sk \equiv H(m) + xr \pmod{q}$$

$$sk - H(m) \equiv xr \pmod{q}$$

$$r^{-1}(sk - H(m)) \equiv x \pmod{q}$$

Again, $H(m)$ is public, plus the attacker needed it to compute $k$, anyway. They've already computed $k$, and $s$ is plucked straight from the signature. That just leaves us with $r^{-1} \pmod{q}$ (read as: "the modular inverse of $r$ modulo $q$"), but that can be computed efficiently as well. (For more information, see the appendix on modular arithmetic; keep in mind that $q$ is prime, so the modular inverse can be computed directly.) That means that the attacker, once they've discovered the $k$ of any signature, can recover the private key directly.

So far, we've assumed that the broken signer would always use the same $k$. To make matters worse, a signer only has to re-use $k$ *once* in any two signatures that the attacker can see for the attack to work. As we've seen, if $k$ is repeated, the $r_i$ values repeat as well. Since $r_i$ is a part of the signature, it's very easy to see when the signer has made this mistake. So, even if reusing $k$ is something the signer only does rarely (because their random number generator is broken, for example), doing it once is enough for the attacker to break the DSA scheme.

In short, reusing the $k$ parameter of a DSA signing operation means an attacker recovers the private key.

TODO: Debian http://rdist.root.org/2009/05/17/the-debian-pgp-disaster-that-almost-was/

## 12.4 ECDSA

TODO: explain (see #53)

As with regular DSA, the choice of $k$ is extremely critical. There are attacks that manage to recover the signing key using a few thousand signatures when only a few bits of the *nonce* leak. [MHMP13]

## 12.5 Repudiable authenticators

Signatures like the ones we described above provide a property called *non-repudiation*. In short, it means that you can't later deny being the sender of the signed message. Anyone can verify that the signature was made using your private key, something only you could do.

That may not always be a useful feature; it may be more prudent to have a scheme where only the intended recipient can verify the signature. An obvious way to design such a scheme would be to make sure that the recipient (or, in fact, anyone else) could have computed an identical value.

Such messages can be repudiated; such a scheme is often called "deniable authentication". While it authenticates the sender to the intended recipient, the sender can later deny (to third parties) having sent the message. Equivalently, the recipient can't convince anyone else that the sender sent that particular message.

13

# Key derivation functions

## 13.1 Description

A key derivation function is a function that derives one or more secret values (the *keys*) from one secret value.

Many key derivation functions can also take a (usually optional) *salt* parameter. This parameter causes the key derivation function to not always return the same output keys for the same input secret. As with other cryptosystems, *salt*s are fundamentally different from the secret input: *salt*s generally do not have to be secret, and can be re-used.

Key derivation functions can be useful, for example, when a cryptographic protocol starts with a single secret value, such as a shared password or a secret derived using Diffie-Hellman key exchange, but requires multiple secret values to operate, such as encryption and MAC keys. Another use case of key derivation functions is in cryptographically secure random number generators, which we'll see in more detail in a following chapter, where they are used to extract randomness with high entropy density from many sources that each have low entropy density.

There are two main categories of key derivation functions, depending on the entropy content of the secret value, which determines how many different possible values the secret value can take.

If the secret value is a user-supplied password, for example, it typically contains very little entropy. There are very few values the password will take. As we've already established in *a previous section on password storage* (page 104), that means it is necessary that the key derivation function is hard to compute. That means it requires a non-trivial amount of computing resources, such as CPU cycles or memory. If the key derivation function were easy to compute, an attacker could simply enumerate all possible values of the shared secret, since there are few possibilities, and then compute the key derivation function for all of them. As we've seen in that previous section on password storage, this is how most modern attacks on password stores work. Using an appropriate key derivation function would prevent these attacks. In this chapter, we'll see scrypt, as well as other key derivation functions in this category.

On the other hand, the secret value could also have a high entropy content. For example, it could be a shared secret derived from a Diffie-Hellman *key agreement* protocol, or an API key consisting of cryptographically random bytes (we'll discuss cryptographically secure random number generation in the next chapter). In that case, it isn't necessary to have a key derivation function that's hard to compute: even if the key derivation function is trivial to compute, there are too many possible values the secret can take, so an attacker would not be able to enumerate them all. We'll see the best-of-breed of this kind of key derivation function, HKDF, in this chapter.

## 13.2 Password strength

TODO: NIST Special Publication 800-63

## 13.3 PBKDF2

## 13.4 bcrypt

## 13.5 scrypt

## 13.6 HKDF

The HKDF, defined in RFC 5869 [KE] and explained in detail in a related paper [Kra10], is a key derivation function designed for high entropy inputs, such as shared secrets from a Diffie-Hellman key exchange. It is specifically *not* designed to be secure for low-entropy inputs such as passwords.

HKDF exists to give people an appropriate, off-the-shelf key derivation function. Previously, key derivation was often something that was done ad hoc for a particular standard. Usually these ad hoc solutions did not have the extra provisions HKDF does, such as *salt*s or the optional info parameter (which we'll discuss later in this section); and that's only in the best case scenario where the KDF wasn't fundamentally broken to begin with.

HKDF is based on HMAC. Like HMAC, it is a generic construction that uses hash functions, and can be built using any cryptographically secure hash function you want.

### A closer look at HKDF

> This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

HKDF consists of two phases. In the first phase, called the *extraction phase*, a fixed-length key is extracted from the input entropy. In the second phase, called the *expansion phase*, that key is used to produce a number of pseudorandom keys.

### The extraction phase

The extraction phase is responsible for extracting a small amount of data with a high entropy content from a potentially large amount of data with a smaller entropy density.

The extraction phase just uses HMAC with a *salt*:

```
def extract(salt, data):
    return hmac(salt, data)
```

The *salt* value is optional. If the *salt* is not specified, a string of zeroes equal to the length of the hash function's output is used. While the *salt* is technically optional, the designers stress its importance, because it makes the independent uses of the key derivation function (for example, in different applications, or with different users) produce independent results. Even a fairly low-entropy *salt* can already contribute significantly to the security of the key derivation function. [KE] [Kra10]

The extraction phase explains why HKDF is not suitable for deriving keys from passwords. While the extraction phase is very good at *concentrating* entropy, it is not capable of *amplifying* entropy. It is designed for compacting a small amount of entropy spread out over a large amount of data into the same amount of entropy in a small amount of data, but is not designed for creating a set of keys that are difficult to compute in the face of a small amount of available entropy. There are also no provisions for making this phase computationally intensive. [KE]

In some cases, it is possible to skip the extraction phase, if the shared secret already has all the right properties, for example, if it is a pseudorandom string of sufficient length, and with sufficient entropy. However, sometimes this should not be done at all, for example when dealing with a Diffie-Hellman shared secret. The RFC goes into slightly more detail on the topic of whether or not to skip this step; but it is generally inadvisable. [KE]

### The expansion phase

In the expansion phase, the random data extracted from the inputs in the extraction phase is expanded into as much data as is required.

The expansion step is also quite simple: chunks of data are produced using HMAC, this time with the extracted secret, not with the public *salt*, until enough bytes are produced. The data being HMACed is the previous output (starting with an empty string), an “info” parameter (by default also the empty string), and a counter byte that counts which block is currently being produced.

```
def expand(key, info=””):
    ”””Expands the key, with optional info.”””
    output = ””
    for byte in map(chr, range(256)):
        output = hmac(key, output + info + byte)
        yield output

def get_output(desired_length, key, info=””):
    ”””Collects output from the expansion step⊠
↪until enough
    has been collected; then returns that output.
↪”””
    outputs, current_length = [], 0
    for output in expand(key, info):
        outputs.append(output)
        current_length += len(output)

        if current_length >= desired_length:
            break
    else:
        # This block is executed when the for⊠
↪loop *isn't*
        # terminated by the ``break`` statement,
↪ which
```

(continues on next page)

(continued from previous page)

```
        # happens when we run out of ``expand`` outputs
        # before reaching the desired length.
        raise RuntimeError(”Desired length too long”)

    return ””.join(outputs)[:desired_length]
```

Like the *salt* in the extraction phase, the “info” parameter is entirely optional, but can actually greatly increase the security of the application. The “info” parameter is intended to contain some application-specific context in which the key derivation function is being used. Like the *salt*, it will cause the key derivation function to produce different values in different contexts, further increasing its security. For example, the info parameter may contain information about the user being dealt with, the part of the protocol the key derivation function is being executed for or the like. [KE]

14

# Random number generators

> The generation of random numbers is too important to be left to chance.
>
> *Robert R. Coveyou*

## 14.1 Introduction

Many cryptographic systems require random numbers. So far, we've just assumed that they're available. In this chapter, we'll go more in depth about the importance and mechanics of random numbers in cryptographic systems.

Producing random numbers is a fairly intricate process. Like with so many other things in cryptography, it's quite easy to get it completely wrong but have everything *look* completely fine to the untrained eye.

There are three categories of random number generation that we'll consider separately:

- True random number generators

- Cryptographically secure pseudorandom number generators
- Pseudorandom number generators

## 14.2 True random number generators

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> *John von Neumann*

John von Neumann, father of the modern model of computing, made an obvious point. We can't expect to produce random numbers using predictable, deterministic arithmetic. We need a source of randomness that isn't a consequence of deterministic rules.

True random number generators get their randomness from physical processes. Historically, many systems have been used for producing such numbers. Systems like dice are still in common use today. However, for the amount of randomness we need for practical cryptographic algorithms, these are typically far too slow, and often quite unreliable.

We've since come up with more speedy and reliable sources of randomness. There are several categories of physical processes that are used for hardware random number generation:

- Quantum processes
- Thermal processes
- Oscillator drift
- Timing events

Keep in mind that not all of these options necessarily generate high-quality, truly random numbers. We'll elaborate further on how they can be applied successfully anyway.

## Radioactive decay

One example of a quantum physical process used to produce random numbers is radioactive decay. We know that radioactive substances will slowly decay over time. It's impossible to know when the next atom will decay; that process is entirely random. Detecting when such a decay has occurred, however, is fairly easy. By measuring the time between individual decays, we can produce random numbers.

## Shot noise

Shot noise is another quantum physical process used to produce random numbers. Shot noise is based on the fact that light and electricity are caused by the movement of indivisible little packets: photons in the case of light, and electrons in the case of electricity.

## Nyquist noise

An example of a thermal process used to produce random numbers is Nyquist noise. Nyquist noise is the noise that occurs from charge carriers (typically electrons) traveling through a medium with a certain resistance. That causes a tiny current to flow through the resistor (or, alternatively put, causes a tiny voltage difference across the resistor).

$$i = \sqrt{\frac{4k_BT\Delta_f}{R}}$$

$$v = \sqrt{4k_BTR\Delta_f}$$

These formulas may seem a little scary to those who haven't seen the physics behind them before, but don't worry too much: understanding them isn't really necessary to go along with the reasoning. These formulas are for the *root mean square*. If you've never heard that term before, you can roughly pretend that means "average". $\Delta f$ is the bandwidth,

$T$ is the temperature of the system in Kelvins, $k_B$ is Boltzmann's constant.

As you can see from the formula, Nyquist noise is *thermal*, or temperature-dependent. Fortunately, an attacker generally can't use that property to break the generator: the temperature at which it would become ineffective is so low that the system using it has probably already failed at that point.

By evaluating the formula, we can see that Nyquist noise is quite small. At room temperature with reasonable assumptions (10 kHz bandwidth and a 1k$\Omega$ resistor), the Nyquist voltage is in the order of several hundred nanovolts. Even if you round up liberally to a microvolt (a thousand nanovolts), that's still a thousandth of a thousandth of a volt, and even a tiny AA battery produces 1.5V.

While the formulas describe the root mean square, the value you can measure will be randomly distributed. By repeatedly measuring it, we can produce high-quality random numbers. For most practical applications, thermal noise numbers are quite high quality and relatively unbiased.

TODO: we've never actually explained the word entropy; "resistance an attacker perceives" is necessary in a good definition

TODO: explain synchronous stream ciphers as CSPRNGs

## 14.3 Cryptographically secure pseudorandom generators

While we'll see several examples of cryptographically secure pseudorandom generators in the next few sections, keep in mind that they are all just algorithms that *could* be used. As an application developer, you should *never* be making a choice between one of them.

Instead, in the few cases you really want to pick a random number manually, you should *always* use the cryptographically secure random number generator provided by your operating system: `/dev/urandom` on *NIX (Linux, BSDs, and OS X), or `CryptGenRandom` on Windows. Python provides handy interfaces to these in the form of `os.urandom` and `random.SystemRandom`.

While they can be implemented securely, try to avoid using userspace cryptographically secure random number generators such as the one in OpenSSL. There are far more things that can go wrong with them, usually involving their internal state: either they remain uninitialized, poorly initialized, or end up re-using the same state in different locations. In all of these cases, the resulting cryptosystem is completely and utterly broken.

TODO: talk about the FUD in the Linux man page for urandom

This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

Since this is a specific cryptographically secure pseudorandom number generator algorithm, you don't actually need to know how it works to write good software. Just use ~urandom~.

## 14.4 Yarrow

This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

The Yarrow algorithm is a cryptographically secure pseudorandom number generator.

TODO: actually explain Yarrow

This algorithm is used as the CSPRNG for FreeBSD, and was inherited by Mac OS X. On both of these operating systems, it's used to implement `/dev/random`. Unlike on Linux, `/dev/urandom` is just an alias for `/dev/random`.

## 14.5 Blum Blum Shub

TODO: explain this, and why it's good (provable), but why we don't use it (slow)

## 14.6 `Dual_EC_DRBG`

> This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

`Dual_EC_DRBG` is a NIST standard for a cryptographically secure pseudorandom bit generator. It sparked a large amount of controversy: despite being put forth as an official, federal cryptographic standard, it quickly became evident that it wasn't very good.

Cryptanalysis eventually demonstrated that the standard could contain a back door hidden in the constants specified by the standard, potentially allowing an unspecified attacker to completely break the random number generator.

Several years afterwards, leaked documents suggested a backdoor in an unnamed NIST standard released in the same year as `Dual_EC_DRBG`, fueling the suspicions further. This led to an official recommendation from the standards body to stop using the standard, which was previously unheard of under such circumstances.

## Background

For a long time, the official standards produced by NIST lacked good, modern cryptographically secure pseudorandom number generators. It had a meager choice, and the ones that had been standardized had several serious flaws.

NIST hoped to address this issue with a new publication called SP 800-90, that contained several new cryptographically secure pseudorandom number generators. This document specified a number of algorithms, based on different cryptographic primitives:

1. Cryptographic hash functions
2. HMAC
3. Block ciphers
4. Elliptic curves

Right off the bat, that last one jumps out. Using elliptic curves for random number generation was unusual. Standards like these are expected to be state-of-the-art, while still staying conservative. Elliptic curves had been considered before in an academic context, but that was a far cry from being suggested as a standard for common use.

There is a second reason elliptic curves seem strange. HMAC and block ciphers are obviously symmetric algorithms. Hash functions have their applications in asymmetric algorithms such as digital signatures, but aren't themselves asymmetric. Elliptic curves, on the other hand, are exclusively used for asymmetric algorithms: signatures, key exchange, encryption.

That said, the choice didn't come entirely out of the blue. A choice for a cryptographically secure pseudorandom number generator with a strong number-theoretical basis isn't unheard of: Blum Blum Shub is a perfect example. Those generators are typically much slower than the alternatives. `Dual_EC_DRBG`, for example, is three orders of magnitude slower than its peers presented in the same standard. The idea is that the extra confidence inspired by the stronger mathematical guarantees is worth the performance penalty. For example, we're fairly confident that factoring numbers is hard, but we're a lot less sure about our hash functions and ciphers. RSA came out in 1977 and has stood the test of time quite well since then. DES came out two years later, and is now considered completely broken. MD4 and MD5 came out over a decade later, and are completely broken as well.

The problem is, though, that the standard didn't actually provide the security proof. The standard specifies the generator but then merely suggests that it would be at least as hard as solving the elliptic curve discrete log problem. Blum Blum Shub, by contrast, has a proof that shows that breaking it is at least as hard as solving the quadratic residuosity problem. The best algorithm we have for that is factoring numbers, which we're fairly sure is pretty hard.

The omission of the proof is a bit silly, because there's no reason you'd use a pseudorandom number generator as slow as `Dual_EC_DRBG` unless you had proof that you were getting something in return for the performance hit.

Cryptographers later did the homework that NIST should have provided in the specification [SS06] [BGjosteen07]. Those analyses quickly highlighted a few issues.

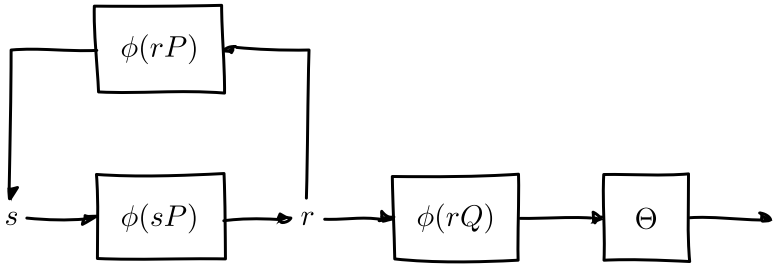
## A quick overview of the algorithm

The algorithm consists of two parts:

1. Generating pseudorandom points on the elliptic curve, which are turned into the internal state of the generator;

2. Turning those points into pseudorandom bits.

We'll illustrate this graphically, with an illustration based on the work by Shumow and Ferguson, two cryptographers who highlighted some of the major issues with this algorithm:



Throughout the algorithm, $\phi$ is a function that takes a curve point and turns it into an integer. The algorithm needs two given points on the curve: $P$ and $Q$. These are fixed, and defined in the specification. The algorithm has an internal state $s$. When producing a new block of bits, the algorithm turns $s$ into a different value $r$ using the $\phi$ function and elliptic curve scalar multiplication with $P$:

$$r = \phi(sP)$$

That value, $r$, is used both for producing the output bits and updating the internal state of the generator. In order to produce the output bits, a different elliptic curve point, $Q$, is used. The output bits are produced by multiplying $r$ with $Q$, and running the result through a transformation $\theta$:

$$o = \theta(\phi(rQ))$$

In order to perform the state update, $r$ is multiplied with $P$ again, and the result is converted to an integer. That integer is used as the new state $s$.

$$s = \phi(rP)$$

## Issues and question marks

First of all, $\phi$ is extremely simple: it just takes the $x$ coordinate of the curve point, and discards the $y$ coordinate. That means that it's quite easy for an attacker who sees the output value of $\phi$ to find points that could have produced that value. In itself, that's not necessarily a big deal; but, as we'll see, it's one factor that contributes to the possibility of a backdoor.

Another flaw was shown where points were turned into pseudorandom bits. The $\theta$ function simply discards the 16 most significant bits. Previous designs discarded significantly more: for 256-bit curves such as these, they discarded somewhere in the range of 120 and 175 bits.

Failing to discard sufficient bits gave the generator a small bias. The next-bit property was violated, giving attackers a better than 50% chance of guessing the next bit correctly. Granted, that chance was only about one in a thousand better than 50%; but that's still unacceptable for what's supposed to be the state-of-the-art in cryptographically secure pseudorandom number generators.

Discarding only those 16 bits has another consequence. Because only 16 bits were discarded, we only have to guess $2^{16}$ possibilities to find possible values of $\phi(rQ)$ that produced the output. That is a very small number: we can simply enumerate all of them. Those values are the outputs of $\phi$, which as we saw just returns the $x$ coordinate of a point. Since we know it came from a point on the curve, we just have to check if our guess is a solution for the curve equation:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

The constants $a, b, p$ are specified by the curve. We've just guessed a value for $x$, leaving only one unknown, $y$. We can solve that quite efficiently. We compute the right hand side and see if it's a perfect square: $y^2 \equiv q \equiv \sqrt{x^3 + ax + b} \pmod{p}$. If it is, $A = (x, \sqrt{q}) = (x, y)$ is a point on the curve. This gives us a number of possible points $A$, one of which is $rQ$ used to produce the output.

This isn't a big deal at face value. To find the state of the algorithm, an attacker needs to find $r$, so they can compute

$s$. They still need to solve the elliptic curve discrete log problem to find $r$ from $rQ$, given $Q$. We're assuming that problem is hard.

Keep in mind that elliptic curves are primitives used for asymmetric encryption. That problem is expected to be hard to solve in general, but what if we have some extra information? What if there's a secret value $e$ so that $eQ = P$?

Let's put ourselves in the shoes of an attacker knowing $e$. We repeat our math from earlier. One of those points $A$ we just found is the $rQ$ we're looking for. We can compute:

$$\phi(eA) \equiv \phi(erQ) \equiv \phi(rP) \pmod{p}$$

That last step is a consequence of the special relationship between $e, P, Q$. That's pretty interesting, because $\phi(rP)$ is exactly the computation the algorithm does to compute $s$, the new state of the algorithm! That means that an attacker that knows $e$ can, quite efficiently, compute the new state $s$ from any output $o$, allowing them to predict all future values of the generator!

This assumes that the attacker knows which $A$ is the *right* $A$. Because only 16 bits were discarded there are only 16 bits left for us to guess. That gives us $2^{16}$ candidate $x$ coordinates. Experimentally, we find that roughly half of the possible $x$ coordinates correspond to points on the curve, leaving us with $2^{15}$ possible curve points $A$, one of which is $rQ$. That's a pretty small number for a bit of computer-aided arithmetic: plenty small for us to try all options. We can therefore say that an attacker that does know the secret value $e$ most definitely can break the generator.

So, we've now shown that if there is a magical $e$ for which $eQ = P$, and you can pick $P$ and $Q$ (and you don't have to explain where you got them from), that you could break the generator. How do you pick such values?

To demonstrate just how possible it is, the researchers started from the NIST curve's $P$ and $p$ values, but came up with their own $Q'$. They did this by starting with $P$, picking a random $d$ (keeping it secret), and setting $Q' = dP$. The trick is that there's an efficient algorithm for computing $e$ in $eQ' = P$ if you know the $d$ in $Q' = dP$. This is the $e$ we need for our earlier attack. When they tried this out, they discovered that in all cases (that is, for many random $d$), seeing 32 bytes of output was enough to determine the state $s$.

All of this, of course, only demonstrates that it is possible for the specified values of $P$ and $Q$ to be special values with a secret back door. It doesn't provide any evidence that the *actual* values have a backdoor in them. However, given that the standard never actually explains *how* they got the magical value for $Q$, it doesn't really inspire a lot of confidence. Typically, cryptographic standards use "nothing-up-my-sleeve" numbers, such as the value of some constant such as $\pi$ or the natural logarithm base, $e$.

If someone does know the backdoor, the consequences are obviously devastating. We've already argued for the necessity of cryptographically secure pseudorandom number generators: having a broken one essentially means that all cryptosystems that use this generator are completely and utterly defeated.

There are two ways one might try to fix this particular algorithm:

- Make the $\theta$ function more complex to invert, rather than just discarding 16 bits. This makes it harder to find candidate points, and hence, harder to perform the attack. One obvious way would be to discard more bits. Another option would be to use a cryptographically secure hash, or a combination of both.

- Generate random $Q$ every time you start the algorithm, possibly by picking a random $d$ and setting $Q = dP$. Of course, $d$ has to be sufficiently large and truly random: if $\theta$ is unchanged, and there are only a few values $d$ can have, the attacker can just perform the above attack for all values of $d$.

Both of these are really just band-aid solutions; it would be a much better idea to just use a different algorithm altogether. These suggestions don't resolve the issue that it's slow, exotic, and now a retracted standard.

### Aftermath

TODO: Talk about RSA guy's comments + snowden leaks

## 14.7 Mersenne Twister

Mersenne Twister is a very common pseudorandom number generator. It has many nice properties, such as high performance, a huge period[1] of $2^{19937} - 1 \approx 4 \cdot 10^{6001}$, and it passes all but the most demanding randomness tests. Despite all of these wonderful properties, it is *not* cryptographically secure.

### An in-depth look at the Mersenne Twister

This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

To demonstrate why Mersenne Twister isn't cryptographically secure, we'll take a look at how the algorithm works. Fortunately, it's not very complex.

The standard Mersenne Twister algorithm operates on an internal state array $S$ consisting of 624 unsigned 32-bit integers, and an index $i$ pointing to the current integer. It consists of three steps:

[1] The period of a pseudorandom number generator is how many random numbers it produces before the entire sequence repeats.

1. An optional initialization function, which produces an initial state from a small random value called a *seed*.
2. A state generation function, which produces a new state from the old state.
3. An extraction function, also called the *tempering* function, that produces a random number from the current element of the state (the element pointed at by the index $i$).

Whenever the extraction function is called, the index to the current integer is incremented. When all of the current elements of the state have been used to produce a number, the state initialization function is called again. The state initialization function is also called right before the first number is extracted.

So, to recap: the state is regenerated, then the extraction function goes over each of the elements in the state, until it runs out. This process repeats indefinitely.

TODO: illustrate

We'll look at each of the parts briefly. The exact workings of them is outside the scope of this book, but we'll look at them just long enough to get some insight into why Mersenne Twister is unsuitable as a cryptographically secure random number generator.

## The initialization function

The initialization function creates an instance of Mersenne Twister's state array, from a small initial random number called a *seed*.

The array starts with the seed itself. Then, each next element is produced from a constant, the previous element, and the index of the new element. Elements are produced until there are 624 of them.

Here's the Python source code:

```
def uint32(n):
    return 0xFFFFFFFF & n

def initialize_state(seed):
    state = [seed]

    for i in range(1, 624):
        prev = state[-1]
        elem = 0x6c078965 * (prev ^ (prev >>⊠
↪30)) + i
        state.append(uint32(elem))

    return state
```

For those of you who haven't worked with Python or its bitwise operators:

- `>>` and `<<` are right-shift and left-shift
- `&` is binary AND: $0\&0 = 0\&1 = 1\&0 = 0$, and $1\&1 = 1$.
- `^` is binary XOR, `^=` XORs and assigns the result to the name on the left-hand side, so `x ^= k` is the same thing as `x = x ^ k`.

REVIEW: Bitwise arithmetic appendix?

## The state regeneration function

The state regeneration function takes the current state and produces a new state. It is called right before the first number is extracted, and every time all 624 elements of the state have been used up.

The Python source code for this function is fairly simple. Note that it modifies the state array in place, instead of returning a new one.

```
def regenerate(s):
    for i in range(624):
        y = s[i] & 0x80000000
```

(continues on next page)

(continued from previous page)

```
        y += s[(i + 1) % 624] & 0x7fffffff

        z = s[(i + 397) % 624]
        s[i] = z ^ (y >> 1)

        if y % 2:
            s[i] ^= 0x9908b0df
```

The `%` in an expression like `s[(i + n) % 624]` means that a next element of the state is looked at, wrapping around to the start of the state array if there is no next element.

The values `0x80000000` and `0x7fffffff` have a specific meaning when interpreted as sequences of 32 bits. `0x80000000` has only the first bit set; `0x7fffffff` has every bit except the first bit set. Because these are bitwise AND'ed together (&), this effectively means that after the first two lines in the loop, `y` consists of the first bit of the current state element and all the subsequent bits of the next element.

## The tempering function

The tempering function is applied to the current element of the state before returning it as the produced random number. It's easier to just show the code instead of explaining how it works:

```
_TEMPER_MASK_1 = 0x9d2c5680
_TEMPER_MASK_2 = 0xefc60000

def temper(y):
    y ^= uint32(y >> 11)
    y ^= uint32((y << 7) & _TEMPER_MASK_1)
    y ^= uint32((y << 15) & _TEMPER_MASK_2)
    y ^= uint32(y >> 18)
    return y
```

It may not be obvious, especially if you're not used to binary arithmetic, but this function is *bijective* or *one-to-one*:

each 32 bit integer input maps to exactly one output, and vice versa: for each 32 bit integer we get as an output there was exactly one 32 bit integer it could have come from. Because it uses right and left shifts, it might look like it throws away data at first glance, and hence can't possibly be reversible. It's true that those shifts throw some bits away, however, the critical operation here is the inline XOR (^=): those shifts are just used to compute masks that the value to be tempered is XOR'd with. The XOR operations themselves are reversible, and because each independent operation is reversible, their composition is too.

Because the tempering function is one-to-one, there is an inverse function: a function that gives you the untempered equivalent of a number. It may not be obvious to you how to construct that function unless you're a bitwise arithmetic wizard, but that's okay; in the worst case scenario we could still brute-force it. Suppose we just try every single 32 bit integer, and remember the result in a table. Then, when we get a result, we look it up in the table, and find the original. That table would have to be at least $2^{32} \cdot 32$ bits in length, or a good 17 gigabytes; big, but not impossibly so.

Fortunately, there's a much simpler method to compute the inverse of the temper function. We'll see why that's interesting when we evaluate the cryptographic security of the Mersenne Twister in the next section. For those interested in the result, the untempering function looks like this:

```python
def untemper(y):
    y ^= y >> 18
    y ^= ((y << 15) & _TEMPER_MASK_2)

    y = _undo_shift_2(y)
    y = _undo_shift_1(y)

    return y

def _undo_shift_2(y):
    t = y
```

(continues on next page)

(continued from previous page)

```
    for _ in range(5):
        t <<= 7
        t = y ^ (t & _TEMPER_MASK_1)

    return t

def _undo_shift_1(y):
    t = y

    for _ in range(2):
        t >>= 11
        t ^= y

    return t
```

## Cryptographic security

Remember that for cryptographic security, it has to be impossible to predict future outputs or recover past outputs given present outputs. The Mersenne Twister doesn't have that property.

It's clear that pseudorandom number generators, both those cryptographically secure and those that aren't, are entirely defined by their internal state. After all, they are deterministic algorithms: they're just trying very hard to pretend not to be. Therefore, you could say that the principal difference between cryptographically secure and ordinary pseudorandom number generators is that the cryptographically secure ones shouldn't leak information about their internal state, whereas it doesn't matter for regular ones.

Remember that in Mersenne Twister, a random number is produced by taking the current element of the state, applying the tempering function, and returning the result. We've also seen that the tempering function has an inverse function. So, if I can see the output of the algorithm and apply the inverse of the tempering function, I've recovered one element out of the 624 in the state.

Suppose that I happen to be the only person seeing the outputs of the algorithm, and you begin at the start of the state, such as with a fresh instance of the algorithm, that means that I can clone the state by just having it produce 624 random numbers.

Even if an attacker doesn't see all 624 numbers, they can often still recreate future states, thanks to the simple relations between past states and future states produced by the state regeneration function.

Again, this is not a weakness of Mersenne Twister. It's designed to be fast and have strong randomness properties. It is not designed to be unpredictable, which is the defining property of a cryptographically secure pseudorandom number generator.

# Part III

# Complete cryptosystems

# 15

# SSL and TLS

## 15.1 Description

SSL, short for Secure Socket Layer, is a cryptographic protocol originally introduced by Netscape Communications[1] for securing traffic on the Web. The standard is now superseded by TLS (Transport Layer Security), a standard publicized in RFCs by the IETF. The term SSL is still commonly used, even when the speaker actually means a TLS connection. From now on, this book will only use the term TLS, unless we really mean the old SSL standard.

Its first and foremost goal is to transport bytes securely, over the Internet or any other insecure medium. [DR] It's a hybrid cryptosystem: it uses both symmetric and asymmetric algorithms in unison. For example, asymmetric algorithms such as signature algorithms can be used to authenticate peers, while public key encryption algorithms or Diffie-Hellman exchanges can be used to negotiate shared secrets and authenticate certificates. On the symmetric side, *stream*

[1] For those too young to remember, Netscape is a company that used to make browsers.

*cipher*s (both native ones and block ciphers in a *mode of operation*) are used to encrypt the actual data being transmitted, and MAC algorithms are used to authenticate that data.

TLS is the world's most common cryptosystem, and hence probably also the most studied. Over the years, many flaws have been discovered in SSL and TLS, despite many of the world's top cryptographers contributing to and examining the standard[2]. As far as we know, the current versions of TLS are secure, or at least can be configured to be secure.

## 15.2 Handshakes

TODO: explain a modern TLS handshake

### Downgrade attacks

SSL 2.0 made the mistake of not authenticating handshakes. This made it easy to mount downgrade attacks. A downgrade attack is a man-in-the-middle attack where an attacker modifies the handshake messages that negotiate which ciphersuite is being used. That way, he can force the clients to set up the connection using an insecure block cipher, for example.

Due to cryptographic export restrictions at the time, many ciphers were only 40 or 56 bit. Even if the attacker couldn't break the best encryption both client and server supported, he could probably break the weakest, which is all that is necessary for a downgrade attack to succeed.

This is one of the many reasons that there is an explicit RFC [TP] prohibiting new TLS implementations from having SSL v2.0 support.

[2] In case I haven't driven this point home yet: it only goes to show that designing cryptosystems is hard, and you probably shouldn't do it yourself.

## 15.3 Certificate authorities

TLS certificates can be used to authenticate peers, but how do we authenticate the certificate? My bank may very well have a certificate claiming to be that particular bank, but how do I know it's actually my bank, and not just someone pretending to be my bank? Why should I trust this particular certificate? As we've seen when we discussed these algorithms, anyone can generate as many key pairs as they'd like. There's nothing stopping someone from generating a key pair pretending to be your bank.

When someone actually tries to use a certificate to impersonate a bank, real browsers don't believe them. They notify the user that the certificate is untrusted. They do this using the standard TLS trust model of certificate authorities. TLS clients come with a list of trusted certificate authorities, commonly shipped with your operating system or your browser. These are special, trusted certificates, that are carefully guarded by their owners.

For a fee, these owners will use their certificate authority to sign other certificates. The idea is that the certificate authority wouldn't sign a certificate for Facebook or a bank or anyone else, unless you could prove you're actually them.

When a TLS client connects to a server, that server provides a certificate chain. Typically, their own certificate is signed by an intermediary CA certificate, which is signed by another, and another, and one that is signed by a trusted root certificate authority. Since the client already has a copy of that root certificate, they can verify the signature chain starting with the root.

Your fake certificate doesn't have a chain leading up to a trusted root certificate, so the browser rejects it.

TODO: Explain why this is a total racket

## 15.4 Self-signed certificates

## 15.5 Client certificates

In TLS, certificates are usually only used to identify the server. This satisfies a typical use case: users want to communicate securely with their banks and e-mail providers, and the certificate authenticates the service they're talking to. The service usually authenticates the user using passwords, and, occasionally, two-factor authentication.

In public-key schemes we've seen so far, all peers typically had one or more key pairs of their own. There's no reason users can't have their own certificates, and use them to authenticate to the server. The TLS specification explicitly supports client certificates. This feature is only rarely used, even though it clearly has very interesting security benefits.

The main reason for that is probably rooted in the poor user experience. There are no systems that rely on client certificates that are easy to use for non-technical people. Since there are few such systems, even tech-savvy people don't know about them, which means new systems aren't created.

Client certificates are a great solution for when you control both ends of the wire and want to securely authenticate both peers in a TLS connection. By producing your own certificate authority, you can even sign these client certificates to authenticate them.

## 15.6 Perfect forward secrecy

Historically, the most common way to agree on the premaster secret is for the client to select a random number and encrypt it, typically using RSA. This has a few nice properties. For example, it means the server can make do with less entropy: since the random bits are handed to the server by the client, the server doesn't need to produce any cryptographically random bits. It also makes the handshake slightly faster, since there's no need for back-and-forth communication to agree on a shared secret.

However, it has one major flaw. Suppose an attacker gets access to the server's private key. Perhaps they managed to factor the modulus of the RSA key, or perhaps they broke in and stole it, or perhaps they used legal force to get the owner to hand over the key. Regardless of how they acquired it, getting access to the key allows the attacker to decrypt all past communication. The key allows them to decrypt the encrypted pre-master secrets, which allows them to derive all of the symmetric encryption keys, and therefore decrypt everything.

There are obvious alternatives to this scheme. We've already seen Diffie-Hellman key exchange, allowing two peers to agree on secret keys over an insecure medium. TLS allows for peers to agree on the pre-master secret using a Diffie-Hellman exchange, either based on discrete logs or elliptic curves.

Assuming both peers discard the keys after use like they're supposed to, getting access to the secret keys wouldn't allow an attacker to decrypt previous communication. That property is called *perfect forward secrecy*. The term "perfect" is a little contested, but the term "forward" means that communications can't be decrypted later if the long-term keys (such as the server's private key) fall into the wrong hands.

Of course, this is only true if Diffie-Hellman exchanges are secure. If an attacker has a significant mathematical and computational advantage over everyone else, such as an algorithm for solving the discrete log problem more efficiently than thought possible, combined with many data centers filled with number-crunching computers, it's possible that they'll break the key exchange itself.

## 15.7 Attacks

As with most attacks, attacks on TLS can usually be grouped into two distinct categories:

1. Attacks on the protocol itself, such as subverting the CA mechanism;

2. Attacks on a particular implementation or cipher, such as cryptanalytic attacks exploiting weaknesses in RC4, or timing attacks in a particular AES implementation.

Unfortunately, SSL/TLS has had many successful attacks in both categories. This section is particularly about the latter.

### CRIME and BREACH

CRIME[3] is an attack by the authors of BEAST. It's an innovative side channel attack that relies on TLS compression leaking information about secrets in the plaintext. In a related attack called BREACH[4], the attackers accomplish the same effect using HTTP compression. That was predicted by the authors of the original paper, but the BREACH authors were the first to demonstrate it as a practical attack. The BREACH attack was more practically applicable, though: HTTP compression is significantly more common than TLS compression.

Both of these rely on encryption of a compressed plaintext, and their mechanisms are virtually identical: only the specific details related to HTTP compression or TLS compression are relevant. The largest difference is that with TLS compression, the entire stream can be attacked; with HTTP compression, only the body is compressed, so HTTP headers are safe. Since the attacks are otherwise extremely similar, we'll just talk about how the attack works in the abstract,

[3] Compression Ratio Info-leak Made Easy

[4] Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext

by explaining how attackers can learn information about the plaintext if it is compressed before encryption.

The most common algorithm used to compress both HTTP and TLS [Hol] is called DEFLATE. The exact mechanics of DEFLATE aren't too important, but the important feature is that byte sequences that occur more than once can be efficiently stored. When a byte sequence recurs[5], instead of recording the same sequence, a reference is provided to the previous sequence: instead of repeating the sequence, it says "go back and look at the thing I wrote N bytes ago".

Suppose an attacker can control the plaintext. For example, the attacker injects an invisible iframe[6] or some JavaScript code that fires off many requests. The attacker needs some way to inject their guess of the secret so that their guess occurs in the plaintext, such as the query parameters[7]. Usually, they can prefix their guess with something known. Suppose they're trying to intercept an authentication token being supplied in the body of the web page:

```
<input type=”hidden”
       name=”csrf-token”
       value=”TOKEN_VALUE_HERE”>
```

... they can prefix the guess with the known part of that. In this case, it's a *CSRF* token; a random token selected by the server and given to the client. This token is intended to prevent malicious third party websites from using the ambient authority present in the browser (such as session cookies) to make authenticated requests. Without a CSRF token, a third party website might just make a request to the vulnerable website; the web browser will provide the stored cookie, and the vulnerable website will mistake that for an authenticated request.

[5] Within limits; specifically within a sliding window, usually 32kB big. Otherwise, the pointers would grow bigger than the sequences they're meant to compress.

[6] An iframe is a web page embedded within a page.

[7] The key-value pairs in a URL after the question mark, e.g. the `x=1&y=2` in `http://example.test/path?x=1&y=2`.

The attacker makes guesses at the value of the token, starting with the first byte, and moving on one byte at a time.[8] When they guess a byte correctly, the ciphertext will be just a little shorter: the compression algorithm will notice that it's seen this pattern before, and be able to compress the plaintext before encrypting. The plaintext, and hence the compressed ciphertext, will therefore be smaller. They can do this directly when the connection is using a *stream cipher* or a similar construction such as *CTR mode*, since they produce ciphertexts that are exactly as long as the plaintexts. If the connection is using a block-oriented mode such as *CBC mode*, the difference might get lost in the block padding. The attacker can solve that by simply controlling the prefix so that the difference in ciphertext size will be an entire block.

Once they've guessed one byte correctly, they can move on to the next byte, until they recover the entire token.

This attack is particularly interesting for a number of reasons. Not only is it a completely new *class* of attack, widely applicable to many cryptosystems, but compressing the plaintext prior to encryption was actively recommended by existing cryptographic literature. It doesn't require any particularly advanced tools: you only need to convince the user to make requests to a vulnerable website, and you only need to be able to measure the size of the responses. It's also extremely effective: the researchers that published BREACH report being able to extract secrets, such as *CSRF* tokens, within one minute.

In order to defend against CRIME, disable TLS compression. This is generally done in most systems by default. In order to defend against BREACH, there are a number of possible options:

- Don't allow the user to inject arbitrary data into the request.

- Don't put secrets in the response bodies.

[8] They may be able to move more quickly than just one byte at a time, but this is the simplest way to reason about.

- Regenerate secrets such as CSRF tokens liberally, for example, each request.

It's a bad idea to simply unconditionally turn off HTTP compression. While it does successfully stop the attack, HTTP compression is a critical tool for making the Web faster.

Web apps that consist of a static front-end (say, using HTML5, JS, CSS) and that only operate using an API, say, JSON over REST, are particularly easy to immunize against this attack. Just disable compression on the channel that actually contains secrets. It makes things slower, of course, but at least the majority of data can still be served over a CDN.

## 15.8 HSTS

HSTS is a way for web servers to communicate that what they're saying should only ever be transferred over a secure transport. In practice, the only secure transport that is ever used for HTTP is TLS.

Using HSTS is quite simple; the web server just adds an extra `Strict-Transport-Security` header to the response. The header value contains a maximum age (`max-age`), which determines how long into the future the browser can trust that this website will be HSTS-enabled. This is typically a large value, such as a year. Browsers successfully remembering that a particular host is HSTS-enabled is very important to the effectiveness of the scheme, as we'll see in a bit. Optionally, the HSTS header can include the `includeSubDomains` directive, which details the scope of the HSTS policy. [HJB]

There are several things that a conforming web browser will do when communicating with an HSTS-enabled website:

- Whenever there is any attempt to make any connection to this website, it will always be done over HTTPS.

The browser does this completely by itself, *before* making the request to the website.

- If there is an issue setting up a TLS connection, the website will not be accessible, instead of simply displaying a warning.

Essentially, HSTS is a way for websites to communicate that they only support secure transports. This helps protect the users against all sorts of attacks including both passive eavesdroppers (that were hoping to see some credentials accidentally sent in plaintext), and active man-in-the-middle attacks such as SSL stripping.

HSTS also defends against mistakes on the part of the web server. For example, a web server might accidentally pull in some executable code, such as some JavaScript, over an insecure connection. An active attacker that can intercept and modify that JavaScript would then have complete control over the (supposedly secure) web site.

As with many TLS improvements, HSTS is not a panacea: it is just one tool in a very big toolbox of stuff that we have to try and make TLS more secure. HSTS only helps to ensure that TLS is actually used; it does absolutely nothing to prevent attacks against TLS itself.

HSTS can suffer from a chicken-or-egg problem. If a browser has never visited a particular HSTS-enabled website before, it's possible that the browser doesn't know that the website is HSTS-enabled yet. Therefore, the browser may still attempt a regular HTTP connection, vulnerable to an SSL stripping attack. Some browsers have attempted to mitigate this issue by having browsers come pre-loaded with a list of HSTS websites.

## 15.9 Certificate pinning

Certificate pinning is an idea that's very similar to HSTS, taken a little further: instead of just remembering that a particular server promises to support HTTPS, we'll remember information about their certificates (in practice, we'll remember a hash of the public key). When we connect to a server that we have some stored information about, we'll verify their certificates, making it much harder for an impostor to pretend to be the website we're connecting to using a different certificate.

Browsers originally implemented certificate pinning by coming shipped with a list of certificates from large, high-profile websites. For example, Google included whitelisted certificates for all of their services in their Chrome browser.

## 15.10 Secure configurations

In this section, we are only talking about configuration options such as which ciphers to use, TLS/SSL versions, etc. We're specifically *not* talking about TLS configurations in the sense of trust models, key management, etc.

There are several issues with configuring TLS securely:

1. Often, the defaults are unsafe, and people are unaware that they should be changed.

2. The things that constitute a secure TLS configuration can change rapidly, because cryptanalysis and practical attacks are continuously improving.

3. Old clients that still need to be supported sometimes mean that you have to hang on to broken configuration options.

A practical example of some of these points coming together is the BEAST attack. That attack exploited weaknesses in CBC ciphersuites in TLSv1.0, which were parts of the default ciphersuite specifications everywhere. Many people recommended defending against it by switching to RC4. RC4 was already considered cryptographically weak, later cryptanalysis showed that RC4 was even more broken than previously suspected. The attack had been known for years before being practically exploited; it was already fixed in TLSv1.1 in 2006, years before the BEAST paper being published. However, TLSv1.1 had not seen wide adoption.

Good advice necessarily changes over time, and it's impossible to do so in a persistent medium such as a book. Instead, you should look at continuously updated third party sources such as Qualys SSL Labs. They provide tests for both SSL clients and servers, and extensive advice on how to improve configurations.

That said, there are certainly some general things we want from a TLS configuration.

TODO: say stuff we generally want from TLS configurations

TODO: `http://tools.ietf.org/html/draft-agl-tls-chacha20poly1305-01`

# 16

# OpenPGP and GPG

## 16.1 Description

OpenPGP is an open standard that describes a method for encrypting and signing messages. GPG is the most popular implementation of that standard[1], available under a free software license.

Unlike TLS, which focuses on data in motion, OpenPGP focuses on data at rest. A TLS session is active: bytes fly back and forth as the peers set up the secure channel. An OpenPGP interaction is, by comparison, static: the sender computes the entire message up front using information shared ahead of time. In fact, OpenPGP doesn't insist that anything is *sent* at all: for example, it can be used to sign software releases.

Like TLS, OpenPGP is a hybrid cryptosystem. Users have key pairs consisting of a public key and a private key. Public key algorithms are used both for signing and encryption. Symmetric key algorithms are used to encrypt the message

[1] GPG 2 also implements S/MIME, which is unrelated to the OpenPGP standard. This chapter only discusses OpenPGP.

body; the symmetric key itself is protected using *public-key encryption*. This also makes it easy to encrypt a message for multiple recipients: only the secret key has to be encrypted multiple times.

## 16.2 The web of trust

Earlier, we saw that TLS typically uses trusted root certificates to establish that a particular peer is who they claim to be. OpenPGP does not operate using such trusted roots. Instead, it relies on a system called the Web of Trust: a friend-of-a-friend honor system that relies on physical meetings where people verify identities.

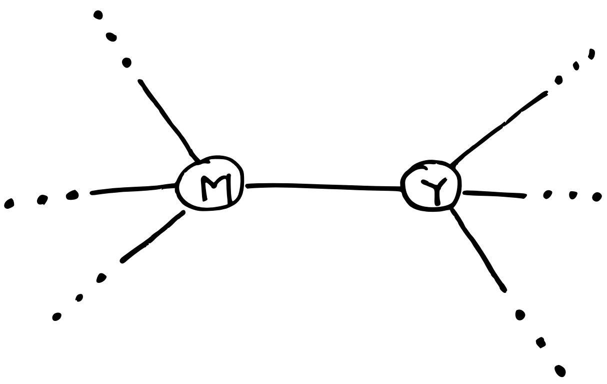
The simplest case is a directly trusted key. If we meet up in person, we can verify each other's identities. Perhaps we know each other, or perhaps we'd check some form of identification. Then, we sign each other's keys.

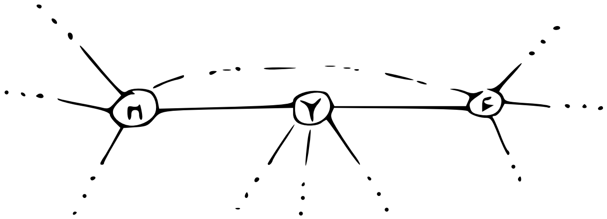
Because I know the key is yours, I know that you can read the messages encrypted by it, and the other way around. Provided you don't share your key, I know that *only* you can read those messages. No-one can replace my copy of your key, because they wouldn't be able to forge my signature on it.

There's a direct trust link between the two of us, and we can communicate securely.

A slightly more complicated case is when a friend of yours would like to send me a message. We've never met: he's never signed my key, nor have I signed theirs. However, I have signed your key, and vice versa. You've signed your friend's key, and vice versa. Your friend can choose to leverage your assertion that I'm indeed the person in possession of that key you signed, and use that to communicate with me securely.

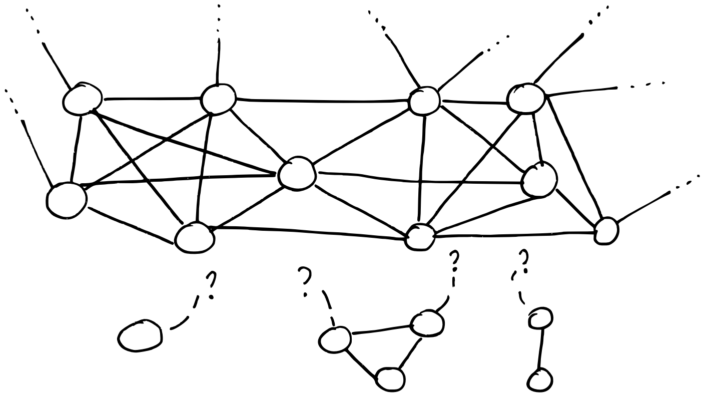
You might wonder how your friend would ever see signatures that you placed on my key. This is because keys and signatures are typically uploaded to a network of key servers, making them freely available to the world.





The above system can be extended to multiple layers of friends. It relies in no small part in communities being linked by signatures, which is why many community events include key signing parties, where people sign each other's keys. For large events, such as international programming conferences, this system is very effective. The main weakness in this system are "islands" of trust: individuals or small groups with no connections to the rest of the web.

Of course, this is only the default way to use OpenPGP. There's nothing stopping you from shipping a particular public key as a part of a software package, and using that to sign messages or verify messages. This is analogous to how you might want to ship a key with a client certificate, or a custom



root CA certificate, with TLS.

# 17

# Off-The-Record Messaging (OTR)

## 17.1 Description

*OTR messaging* is a protocol for securing instant messaging communication between people [BGB04]. It intends to be the online equivalent of a private, real-life conversation. It encrypts messages, preventing eavesdroppers from reading them. It also authenticates peers to each other, so they know who they're talking to. Despite authenticating peers, it is designed to be deniable: participants can later deny to third parties anything they said to each other. It is also designed to have perfect forward secrecy: even a compromise of a long-term public key pair doesn't compromise any previous conversations.

The deniability and perfect forward secrecy properties are very different from those of other systems such as OpenPGP. OpenPGP intentionally guarantees non-repudiability. It's a great property if you're signing software packages, talking on mailing lists or signing business invoices, but the authors of *OTR* argue that those aren't desirable properties for the online equivalent of one-on-one conversations. Furthermore, OpenPGP's static model of communication makes the constant key renegotiation to facilitate *OTR*'s perfect forward secrecy impossible.

*OTR* is typically configured opportunistically, which means that it will attempt to secure any communication between two peers, if both understand the protocol, without interfering with communication where the other peer does not. The protocol is supported in many different instant messaging clients either directly, or with a plugin. Because it works over instant messages, it can be used across many different instant messaging protocols.

A peer can signal that they would like to speak *OTR* with an explicit message, called the *OTR* Query message. If the peer is just willing to speak *OTR* but doesn't require it, they can optionally invisibly add that information to a plaintext message. That happens with a clever system of whitespace tags: a bunch of whitespace such as spaces and tab characters are used to encode that information. An *OTR*-capable client can interpret that tag and start an *OTR* conversation; an client that isn't *OTR*-capable just displays some extra whitespace.

*OTR* uses many of the primitives we've seen so far:

- Symmetric key encryption (AES in CTR mode)
- *Message authentication codes* (HMAC with SHA-1)
- Diffie-Hellman key exchange

*OTR* also utilizes another mechanism, called the SMP, to check if peers arrived at the same shared secret.

## 17.2 Key exchange

In *OTR*, AKE relies heavily on Diffie-Hellman key exchange, extended with a significant number of extra, interlocking checks. The Diffie-Hellman exchange itself uses a fixed 1536-bit prime with a fixed generator $g$.

We suppose that two participants, named Alice and Bob want to communicate and are willing to exchange sensitive data with each other. Alice and Bob have a long-term DSA authentication key pair each, which we'll call $(p_A, s_A)$ and $(p_B, s_B)$ respectively.

The protocol also relies on a number of other primitives:

- A 128-bit block cipher. In *OTR*, this is always AES. In this section, we'll call block cipher encryption and decryption $E$ and $D$, respectively.
- A hash function, $H$. In *OTR*, this is SHA1.
- A *message authentication code*, $M$. In *OTR*, this is HMAC-SHA1.
- A signing function, $S$.

## Commit message

Initially Alice and Bob are in a protocol state where they wait for the peer to initiate an *OTR* connection, and advertise their own capability of speaking *OTR*.

Let's suppose that Bob chooses to initiate an *OTR* conversation with Alice. His client sends an *OTR* Commit Message, and then transitions to a state where he waits for a reply from from Alice's client.

To send a commit message, a client picks a random 128-bit value $r$ and a random 320-bit (or larger) Diffie-Hellman secret $x$. It then sends $E(r, g^x)$ and $H(g^x)$ to the peer.

## Key message

Alice's client has received Bob's client's advertisement to start an *OTR* session. Her client replies with a key message, which involves creating a new Diffie-Hellman key pair. She picks a 320-bit (or larger) Diffie-Hellman secret $y$ and sends $g^y$ to Bob.

## Reveal Signature Message

Now that Alice has sent her public Diffie-Hellman key, Bob can complete his part of the Diffie-Hellman protocol. Alice can't continue yet, because she hasn't seen Bob's public key.

When we discussed Diffie-Hellman, we noted that it does not *authenticate* the peer. Bob can compute a secret, but doesn't know he's talking to Alice. As with TLS and other systems using Diffie-Hellman, this problem is solved by authenticating the key exchange.

After verifying that Alice's public key is a valid value, Bob computes the shared secret $s = (g^y)^x$. Using a key derivation function, he derives several keys from $s$: two AES keys $c, c'$, and four MAC keys $m_1, m_1', m_2, m_2'$.

He chooses an identification number $i_B$ for his current Diffie-Hellman key pair $(x, g^x)$. This will be important once Alice and Bob generate new key pairs, which they will do later on in the *OTR* protocol.

Bob computes:

$$M_B = M_{m_1}(g^x, g^y, p_B, i_B)$$

$$X_B = (p_B, i_B, S(p_B, M_B))$$

He sends Alice $r, E_c(X_B), M_{m_2}(E_c(X_B))$.

## Signature Message

Alice can now confirm she's talking to Bob directly, because Bob signed the authenticator for the exchange $M_B$ with his long-term DSA key.

Alice can now also compute the shared secret: Bob has sent her $r$, which was previously used to encrypt Bob's Diffie-Hellman public key. She then computes $H(g^x)$ herself, to compare it against what Bob sent. By completing her side of the Diffie-Hellman exchange ($s = (g^x)^y$), she derives the same keys: $c, c', m_1, m_1', m_2, m_2'$. Using $m_2$, she can verify $M_{m_2}(E_c(X_B))$. Once that message is verified, she can safely decrypt it using her computed $c$.

She can then also compute $M_B = M_{m_1}(g^x, g^y, p_B, i_B)$, and verifies that it is the same as Bob sent. By verifying the signed portion $S(p_B, M_B)$ against Bob's public key, she has now unambiguously tied the current interaction to Bob's long-term authentication key.

She then computes the same values Bob computed to tie his long-term key to the short-term handshake, so that Bob can also authenticate her. She chooses an identification number $i_A$ for her current DH keypair $(y, g^y)$, computes $M_A = M_{m'_1}(g^y, g^x, p_A, i_A)$ and $X_A = p_A, i_A, S(p_A, M_A)$. Finally, she sends Bob $E_{c'}(X_A), M_{m'_2}(E_c(X_B))$.

## Authenticating Alice

Now Bob can also authenticate Alice, again by mirroring steps. First, he verifies $M_{m'_2}(E_c(X_B))$. This allows him to check that Alice saw the same $X_B$ he sent.

Once he decrypts $E_{c'}(X_A)$, he has access to $X_A$, which is Alice's long-term public key information. He can then compute $M_A = M_{m'_1}(g^y, g^x, p_A, i_A)$ to compare it with the version Alice sent. Finally, he verifies $S(p_A, M_A)$ with Alice's public key.

## What have we accomplished?

If all checks succeed then Alice and Bob have completed an authenticated Diffie-Hellman exchange and have a shared secret that only the two of them know.

Now that you've seen both sides of the authenticated handshake, you can see why so many different keys are derived from the Diffie-Hellman secret. Keys marked with a prime (′) are for messages originating from the second peer (the one responding to the advertisement, in our case, Alice); keys without a prime are for the initiating peer (in our case, Bob).

## 17.3 Data exchange

TODO: Explain (`https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html`), #33

# Part IV

# Appendices

A

# Modular arithmetic

Modular arithmetic is used for many public key cryptosystems, including *public-key encryption* algorithms like RSA and key exchange protocols like Diffie-Hellman.

Modular arithmetic is something most people actually already understand, they just don't know it's called that. We can illustrate the principles of modular arithmetic using a clock.

For simplicity's sake, our demonstration 12-hour clock only shows hours, not minutes or seconds. Also unlike real clocks, the hour hand is never halfway in between two hours: it always shows an exact hour, such as 2 or 9.

## A.1 Addition and subtraction

It obviously makes sense to add hours on our clock: if it's 2 o'clock now, and you'd like to know what time it is five hours from now, you can add 5, and end up with 7, as you can see in Figure 1.2.

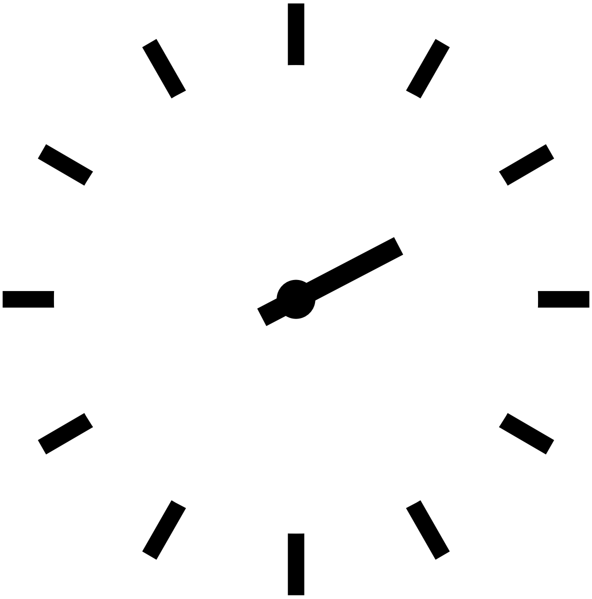


Figure 1.1: A clock, pointing to 2.

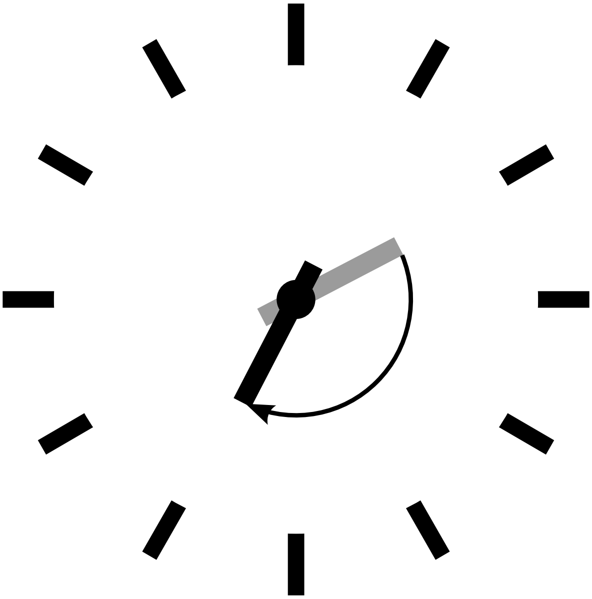


Figure 1.2: $2 + 5 = 7$, on the clock.

Similarly, we can subtract times. If it's 10 o'clock now, and you'd like to know what time it was two hours ago, you subtract 2 and end up with 8.

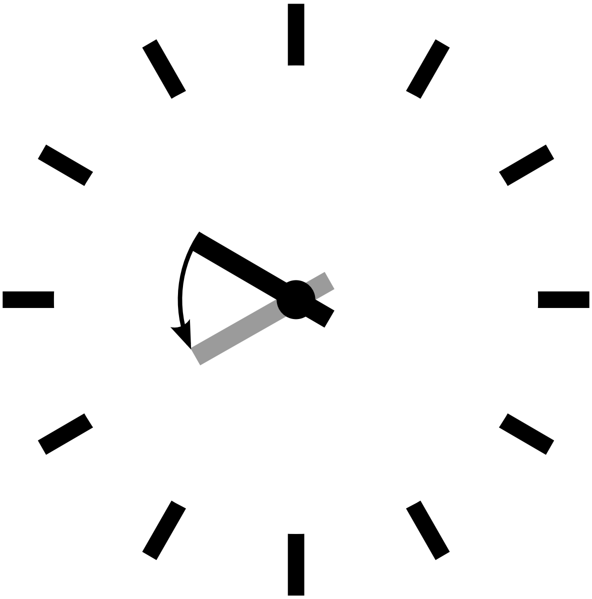


Figure 1.3: $10 - 2 = 8$, on the clock.

The "weird" part is when you cross the boundary at 12. As far as the clock is concerned, there's no real difference between 12 and 0. If it's 10 o'clock now, it'll be 2 o'clock in four hours. If it's 2 o'clock now, it was 9 o'clock five hours ago.

This is an example of what's called "modular arithmetic". The modulus, in this case, is 12. We can write the above equations as:

$$(10 + 4) \bmod 12 = 2$$

$$(2 - 5) \bmod 12 = 9$$

In these equations, the mod is an operator, giving the remainder after division. When we are dealing with modular arithmetic, where all operations are affected by the modulus instead of a simple single operation, we'll instead write $(\text{mod } 12)$ at the end of the equation and use an $\equiv$ sign instead of an equals sign ($=$):

$$10 + 4 \equiv 2 \pmod{12}$$

$$2 - 5 \equiv 9 \pmod{12}$$

This is read as "ten plus four is equivalent to two, modulo twelve" and "two minus five is equivalent to nine, modulo twelve". That might seem like a trivial notational hack now, but the difference will become apparent once we start applying tricks for doing more complex modular computations, like multiplication and exponentiation.

In general, we call two numbers *equivalent modulo some modulus* if dividing them by the modulus leaves the same remainder. We can illustrate this with our previous examples: $10 + 4 = 14$ leaves a remainder of 2 when divided by 12, so it is equivalent to 2 modulo 12. For negative numbers, we'll always use positive remainders. For example, $2 - 5 \equiv 9 \pmod{12}$. This is exactly the way a clock works as well: if it's 2 o'clock now, then five hours ago was "nine o'clock", not "minus three o'clock".

## A.2 Prime numbers

Prime numbers are wonderful kinds of numbers that come back in many branches of mathematics. Anything I say about them probably won't do them justice; but we're in a practical book about applied cryptography, so we'll only see a few properties.

A prime number is a number that is divisible only by two numbers: 1 and itself. For example, 3 is a prime number, but 4 is not, because it can be divided by 2.

Any number can be written as a product of prime factors: a bunch of prime numbers multiplied together. That product is called a prime factorization. For example, 30 can be factorized into 2, 3 and 5:

$$30 = 2 \cdot 3 \cdot 5$$

Sometimes, a prime number will occur more than once in a factorization. For example, the factorization of 360 has 2 in it three times, and three in it twice:

$$360 = 2^3 \cdot 3^2 \cdot 5$$

The factorization of any prime number is just that prime number itself.

Modern mathematics no longer considers 1 to be a prime number, even though it is only divisible by 1 and itself (1 again). Under this convention, every number not only *has* a factorization, but that factorization is *unique*. Otherwise, 4 could be factored not only as $2 \cdot 2$, but also as $2 \cdot 2 \cdot 1$, $2 \cdot 2 \cdot 1 \cdot 1$, and so on. The uniqueness of factorization helps in some important proofs in number theory.

Also, 0 is *not* a prime number, as it is divisible by many numbers: all numbers except 0 itself.

Two numbers are called coprime when their greatest common divisor is 1, or, to put it in another way, they don't share any prime factors. Since the only prime factor a prime has is itself, that means that all prime numbers are also coprime. More generally, a prime is coprime to any number that isn't a multiple of that prime.

## A.3 Multiplication

You might remember you were first taught multiplication as repeated addition:

$$n \cdot x = \underbrace{x + x + \ldots + x}_{n \text{ times}}$$

Modular multiplication is no different. You can compute modular multiplication by adding the numbers together, and taking the modulus whenever the sum gets larger than the modulus. You can also just do regular multiplication, and then take the modulus at the end.

## A.4 Division and modular inverses

Division is defined as the inverse of multiplication. So, $a \cdot b \equiv c \pmod{m}$, then $\frac{c}{b} \equiv a \pmod{m}$.

For example, $5 \cdot 6 \equiv 2 \pmod{7}$; so: $\frac{2}{6} \equiv 5 \pmod{7}$. This is because $5 \cdot 6 = 30$, which leaves a remainder of 2 when divided by 7.

Usually, instead of using division directly, we'll multiply using something called a modular inverse. The modular inverse of $a$ is a number, that when you multiply it with $a$, you get 1. This is just like the inverse of a number in regular arithmetic: $x \cdot \frac{1}{x} = 1$.

Like in regular arithmetic, not all numbers have modular inverses. This is the equivalent of dividing by zero in regular arithmetic.

There are two algorithms that are used to compute modular inverses: the extended Euclidean algorithm, and with the help of Euler's theorem.

### The extended Euclidean theorem

TODO: explain, and how you can get modular inverses with it

### Using Euler's theorem

Euler's theorem states that if two numbers $a$ and $n$ are coprime, then:

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

In that equation, $\phi$ is Euler's totient function, which counts the amount of numbers that are coprime to (and less than or equal to) its argument. As an example, the totient of 10 is 4, as 1, 3, 7, and 9 do not have common prime factors with 10.

We can use Euler's theorem to find the multiplicative inverse of $a$. If we just multiply both sides of the equation by $a^{-1}$, we get:

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

That gives us a direct formula for computing $a^{-1}$. Unfortunately, this is still generally less interesting than using the extended Euclidean algorithm, for two reasons:

1. It requires computing the totient function, which is harder than running the extended Euclidean algorithm in the first place, unless you happen to know the prime factors of $n$.

2. Modular exponentiation is computationally expensive.

One exception to that rule is for prime moduli. Since a prime is coprime to every other number, and since there are $p - 1$ numbers smaller than $p$, $\phi(p) = p - 1$. So, for a prime modulus, the modular inverse of $a$ is simply:

$$a^{-1} \equiv a^{\phi(p)-1} \equiv a^{p-2} \pmod{p}$$

This still requires us to be able to efficiently raise $a$ to a power using modular arithmetic. We'll discuss how you can do that efficiently in the next section.

## A.5 Exponentiation

Like multiplication is taught as repeated addition, exponentiation can be thought of as repeated multiplication:

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ times}}$$

As with multiplication, it's possible to compute modular exponentiation by performing regular exponentiation, and then taking the modulus at the end. However, this is very inefficient, particularly for large $n$: the product quickly becomes far too large.

Fortunately, it is possible to compute modular exponentiation much more efficiently. This is done by splitting the problem up into smaller sub-problems. For example, instead of computing $2^{20}$ directly you could split it up:

$$2^{20} = (2^{10})^2$$

$2^{10}$ is something you can compute on your hands: start at 2, which is $2^1$, and then keep multiplying by two. Every time you multiply by two, the exponent goes up by 1, so by the time you've counted all your fingers (assuming you have ten of them), you're done. The result is 1024. So:

$$\begin{aligned} 2^{20} &\equiv (2^{10} \bmod 15)^2 \pmod{15} \\ &\equiv (1024 \bmod 15)^2 \pmod{15} \\ &\equiv 4^2 \pmod{15} \\ &\equiv 16 \pmod{15} \\ &\equiv 1 \pmod{15} \end{aligned}$$

## A.6 Exponentiation by squaring

A particularly efficient way to do it on computers is splitting the exponent up into a sum of powers of two. This is called exponentiation by squaring, or sometimes also binary exponentiation. Suppose we want to compute $3^{209} \pmod{19}$. First, we split up 209 into a sum of powers of two. This process is essentially just writing 209 down in binary: `0b11010001`. That's very practical if the computation is being performed by a computer, because that's typically how the computer had the number stored in the first place.

$$\begin{aligned}209 &= 1\cdot 2^7 &&+1\cdot 2^6 &&+0\cdot 2^5 &&+1\cdot 2^4 &&+0\cdot 2^3 &&+0\cdot 2^2 &&+0\cdot 2^1 &&+1\cdot 2^0\\ &= 1\cdot 128 &&+1\cdot 64 &&+0\cdot 32 &&+1\cdot 16 &&+0\cdot 8 &&+0\cdot 4 &&+0\cdot 2 &&+1\cdot 1\\ &= 128 &&+64 && &&+16 && && && &&+1\end{aligned}$$

We use that expansion into a sum of powers of two to rewrite the equation:

$$\begin{aligned}3^{209} &= 3^{128+64+16+1}\\ &= 3^{128}\cdot 3^{64}\cdot 3^{16}\cdot 3^{1}\end{aligned}$$

Now, we need to compute those individual powers of 3: 1, 16, 64 and 128. A nice property of this algorithm is that we don't actually have to compute the big powers separately from scratch. We can use previously computed smaller powers to compute the larger ones. For example, we need both $3^{128} \pmod{19}$ and $3^{64} \pmod{19}$, but you can write the former in terms of the latter:

$$3^{128} \bmod 19 = (3^{64} \bmod 19)^2 \pmod{19}$$

Let's compute all the powers of 3 we need. For sake of brevity, we won't write these out entirely, but remember that all tricks we've already seen to compute these still apply:

$$\begin{aligned}3^{16} &\equiv 17 \pmod{19}\\ 3^{64} &\equiv (3^{16})^4 \equiv 17^4 \equiv 16 \pmod{19}\\ 3^{128} &\equiv (3^{64})^2 \equiv 16^2 \equiv 9 \pmod{19}\end{aligned}$$

Filling these back in to our old equation:

$$\begin{aligned}3^{209} &= 3^{128}\cdot 3^{64}\cdot 3^{16}\cdot 3^{1} \pmod{19}\\ &\equiv 9\cdot 16\cdot 17\cdot 3 \pmod{19}\end{aligned}$$

This trick is particularly interesting when the exponent is a very large number. That is the case in many cryptographic applications. For example, in RSA decryption, the exponent is the private key $d$, which is usually more than a thousand bits long. Keep in mind that this method will still leak timing information, so it's only suitable for offline computation. Modular exponentiation can also be computed using a technique called a Montgomery ladder, which we'll see in the next section.

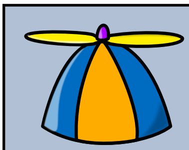
Many programming languages provide access to specific modular exponentiation functions. For example, in Python, `pow(e, x, m)` performs efficient modular exponentiation. However, the expression `(e ** x) % m` will still use the inefficient method.

## A.7 Montgomery ladder exponentiation

As we mentioned before, the exponentiation by squaring algorithm is simple and fast, but the time it takes to complete depends on the value of the exponent. That's bad, because the exponent is usually a secret value, such as a Diffie-Hellman secret or the private exponent $d$ in RSA.

The Montgomery ladder is an algorithm that resolves this by guaranteeing the same number of operations irrespective of the particular value of the exponent. It was originally applied for efficient scalar multiplications over elliptic curves, but the mathematics works for many other systems: specifically, for any abelian group. [JY02]

### Deriving the ladder



This is an optional, in-depth section. It almost certainly won't help you write better software, so feel free to skip it. It is only here to satisfy your inner geek's curiosity.

This section involves a good deal of arithmetic tricks. You might want to get out some paper and pencil to follow along.

Like with exponentiation by squaring, we start by looking at the binary expansion of the exponent $k$. Generally, any $k$ can be written as a sum ($\sum$) of some powers of two ($2^i$). If $2^j$ appears in the binary expansion, we'll say that $k_j = 1$; if it doesn't, we'll say that $k_j = 0$. That gives us:

$$k = \sum_{i=0}^{t-1} 2^i k_i$$

That definition might look scary, but all you're really doing here is defining $k_i$ as bit of $k$ at position $i$. The sum goes over all the bits: if $k$ is $t$ bits long, and we start indexing at 0, the index of the highest bit is $t - 1$, and the index of the lowest bit is 0. For example, the binary expansion of the number 6 is `0b110`. That number is three bits long, so $t = 3$. So:

$$\begin{aligned} 6 &= \sum_{i=0}^{t-1} 2^i k_i \\ &= \sum_{i=0}^{2} 2^i k_i \\ &= k_2 \cdot 2^2 + k_1 \cdot 2^1 + k_0 \cdot 2^0 \\ &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \end{aligned}$$

So, $(k_2, k_1, k_0) = (1, 1, 0)$.

The next few steps don't make a lot of sense until you see them come together at the end, so bear with me and check that the math works out. We'll define a related sum, $L_j$:

$$L_j = \sum_{i=j}^{t-1} 2^{i-j} k_i$$

For example, $L_1$ (still with $k = 6$) becomes:

$$\begin{aligned} L_1 &= \sum_{i=1}^{2} 2^{i-1} k_i \\ &= \underbrace{2^1 \cdot k_2}_{i=2} + \underbrace{2^0 \cdot k_1}_{i=1} \\ &= 2 \cdot 1 + 1 \cdot 1 \\ &= 3 \end{aligned}$$

Essentially, $L_j$ is just $k$ shifted to the right by $j$ bits. Shifting to the right by one bit is the same thing as flooring division by two, just like right-shifting by a decimal digit is the same thing as flooring division by 10. For example: 73, shifted one decimal digit to the right is 7; 0b101 (5) shifted one binary digit (bit) to the right is 0b10 (2). Analogously, shifting left is the inverse operation, and is equivalent to *multiplying* by two.

Next, we'll perform a little arithmetical hocus pocus. First of all:

$$L_j = 2 \cdot L_{j+1} + k_j$$

While you can verify this arithmetically, the easiest way to check this is to think of it in terms of right and left shifts. If you shift $k$ to the right by $j$ positions, that

$$\begin{aligned} k &= \texttt{0b110010111} \\ L_j = L_2 &= \texttt{0b1100101} \\ L_{j+1} = L_3 &= \texttt{0b110010} \\ 2 \cdot L_{j+1} = 2 \cdot L_3 &= \texttt{0b1100100} \end{aligned}$$

You can visually verify that $L_2$ is indeed $L_3$, shifted one to the left (which is the same thing as multiplying by two), plus that one bit $k_j$ that "fell off" when shifting right. $k_j$ is the last bit of $L_j$; in this case it happens to be 1, but it could equally well have been 0.

We define another very simple function $H_j$:

$$H_j = L_j + 1 \iff L_j = H_j - 1$$

Starting from our previous result:

$$\begin{aligned} L_j &= 2 \cdot L_{j+1} + k_j \\ &\Downarrow (L_{j+1} = H_{j+1} - 1) \\ L_j &= L_{j+1} + k_j + H_{j+1} - 1 \\ &\Downarrow (L_{j+1} = H_{j+1} - 1) \\ L_j &= 2 \cdot H_{j+1} + k_j - 2 \end{aligned}$$

We can combine these to produce an inductive way to compute $L_j$ and $H_j$:

$$L_j = \begin{cases} 2L_{j+1} & \text{if } k_j = 0, \\ L_{j+1} + H_{j+1} & \text{if } k_j = 1. \end{cases}$$

$$H_j = \begin{cases} L_{j+1} + H_{j+1} & \text{if } k_j = 0, \\ 2H_{j+1} & \text{if } k_j = 1. \end{cases}$$

Remember that we're doing this to compute $g^k$. Let's write the exponentiation out:

$$g^{L_j} = \begin{cases} g^{2L_{j+1}} = \left(g^{L_{j+1}}\right)^2 & \text{if } k_j = 0, \\ g^{L_{j+1}+H_{j+1}} = g^{L_{j+1}} \cdot g^{H_{j+1}} & \text{if } k_j = 1. \end{cases}$$

$$g^{H_j} = \begin{cases} g^{L_{j+1}+H_{j+1}} = g^{L_{j+1}} \cdot g^{H_{j+1}} & \text{if } k_j = 0, \\ g^{2H_{j+1}} = \left(g^{H_{j+1}}\right)^2 & \text{if } k_j = 1. \end{cases}$$

Remember that $L_j$ is $k$ right-shifted by $j$ bits, so $L_0$ is $k$ shifted right by 0 bits, or just $k$ itself. That means $g^k$, the number we're trying to compute, is the same thing as $g^{L_0}$. By starting at $g^{L_{t-1}}$ ($g$ raised to the power of the leftmost bit of $k$) and iteratively making our way down to $g^{L_0} = g^k$, we have an elegant inductive method for computing $g^k$ based on two simple recursive rules.

The important part about this algorithm is the constant number of operations. If $k_j = 0$, computing $g^{L_j}$ involves one squaring and $g^{H_j}$ involves one multiplication; if $k_j = 1$, it's the other way around. No matter what any of the bits of $k$ are, you need one squaring operation and one multiplication per bit.

## Implementing the Montgomery ladder in Python

The Python implementation of this algorithm, applied to modular exponentiation, is surprisingly terse:

```
def montgomery(x, exponent, modulus):
    x1, x2 = x, x ** 2
    high_bit, *remaining_bits = bits(exponent)
    for bit in remaining_bits:
        if bit == 0:
            x2 = x1 * x2
            x1 = x1 ** 2
        else:
            x1 = x1 * x2
            x2 = x2 ** 2
        x1, x2 = x1 % modulus, x2 % modulus
    return x1
```

This code block doesn't show the definition of `bits`: it produces the binary expansion of its argument. Python doesn't provide that by default; `bin` is close, but that produces a string: `bin(100)` evaluates to `0b1100100`. The `a, *b = bits(...)` construct assigns the first item in `bits(...)` to a, and all remaining bits to b, effectively just skipping the first bit.

The important thing to note here is that no matter what the particular value of the exponent is, there is one squaring, one multiplication, and one modulo operation per bit. Keep in mind that this doesn't necessarily make the entire algorithm take constant time, because the individual squaring and multiplication operations are not necessarily constant time.

## A.8 Discrete logarithm

Just like subtraction is the inverse of addition, and division is the inverse of multiplication, logarithms are the inverse of exponentiation. In regular arithmetic, $b^x = y$, if $x = \log_b y$. This is pronounced "$b$ raised to the power $x$ is $y$", and "the logarithm of $y$ with respect to $b$ is $x$". The equivalent of this in modular arithmetic is called a "discrete logarithm".

As with division, if you start from the definition as the inverse of a different operator, it's easy to come up with examples. For example, since $3^6 \equiv 9 \pmod{15}$, we can define $6 \equiv \log_3 9 \pmod{15}$. Unlike modular inverses, computing discrete logarithms is generally hard. There is no formal proof that computing discrete logarithms is *intrinsically* complex; we just haven't found any efficient algorithms to do it. Because this field has gotten extensive research and we still don't have very fast general algorithms, we consider it safe to base the security of protocols on the assumption that computing discrete logs is hard.

There is one theoretical algorithm for computing discrete logarithms efficiently. However, it requires a quantum computer, which is a fundamentally different kind of computer from the classical computers we use today. While we can build such computers, we can only build very small ones. The limited size of our quantum computers strongly limits which problems we can solve. So far, they're much more in the realm of the kind of arithmetic a child can do in their head, than ousting the top of the line classical computers from the performance throne.

The complexity of computing discrete logarithms, together with the relative simplicity of computing its inverse, modular exponentiation, is the basis for many public key cryptosystems. Common examples include the RSA encryption primitive, and the Diffie-Hellman key exchange protocol.

While cryptosystems based on the discrete logarithm problem are currently considered secure with appropriate parameter choices, there are certainly ways that could change in the future. For example:

- Theoretical breakthroughs in number theory could make discrete logarithms significantly easier to compute than we currently think.
- Technological breakthroughs in quantum computing could lead to large enough quantum computers.
- Technological breakthroughs in classical computing as well as the continuous gradual increases in performance and decreases in cost could increase the size of some problems that can be tackled using classical computers.

Discrete logarithm computation is tightly linked to the problem of number factorization. They are still areas of active mathematical research; the links between the two problems are still not thoroughly understood. That said, there are many similarities between the two:

- Both are believed to be hard to compute on classical computers, but neither has a proof of that fact.
- They can both be efficiently computed on quantum computers using Shor's algorithm.
- Mathematical advances in one are typically quickly turned into mathematical advances in the other.

## A.9 Multiplicative order

Given integer $a$ and positive integer $b$ with $\gcd(a, b) = 1$, the *multiplicative order* of $a \pmod{b}$ is the smallest positive integer $k$ such that $a^k = 1 \pmod{b}$.

# B

# Elliptic curves

Like modular arithmetic, elliptic curve arithmetic is used for many public key cryptosystems. Many cryptosystems that traditionally work with modular arithmetic, such as Diffie-Hellman and DSA, have an elliptic curve counterpart.

Elliptic curves are curves with the following form:

$$y^2 = x^3 + ax + b$$

This is called the “short Weierstrass form”, and is the most common form when talking about elliptic curves in general. There are several other forms which mostly have applications in cryptography, notably the Edwards form:

$$x^2 + y^2 = 1 + dx^2y^2$$

We can define addition of points on the curve.

TODO: Move the Abelian group thing somewhere else, since it applies to our fields thing as well

All of this put together form something called an Abelian group. That’s a scary-sounding mathematical term that almost everyone already understands the basics of. Specifically, if you know how to add integers $(\ldots - 2, -1, 0, 1, 2, \ldots)$

together, you already know an Abelian group. An Abelian group satisfies five properties:

1. If $a$ and $b$ are members of the Abelian group and $\star$ is the operator, then $a \star b$ is also a member of that Abelian group. Indeed, any two integers added together always get you another integer. This property is called *closure,* or, we say that the group is *closed under addition* (or whatever the name is of the operation we've defined).

2. If $a$, $b$ and $c$ are members of the Abelian group, the order of operations doesn't matter; to put it differently: we can move the brackets around. In equation form: $(a \star b) \star c = a \star (b \star c)$. Indeed, the order in which you add integers together doesn't matter; they will always sum up to the same value. This property is called *associativity,* and the group is said to be *associative.*

3. There's exactly one identity element $i$, for which $a \star i = i \star a = a$. For integer addition, that's zero: $a + 0 = 0 + a = a$ for all a.

4. For each element $a$, there's exactly one inverse element $b$, for which $a \star b = b \star a = i$, where $i$ is the identity element. Indeed, for integer addition, $a + (-a) = (-a) + a = 0$ for all a.

5. The order of elements doesn't matter for the result of the operation. For all elements $a, b$, $a \star b = b \star a$. This is known as *commutativity,* and the group is said to be *commutative.*

The first four properties are called group properties and make something a group; the last property is what makes a group Abelian.

We can see that our elliptic curve, with the point at infinity and the addition operator, forms an Abelian group:

1. If $P$ and $Q$ are two points on the elliptic curve, then $P + Q$ is also always a point on the curve.

2. If $P$, $Q$, and $R$ are all points on the curve, then $P+(Q+R) = (P+Q)+R$, so the elliptic curve is associative.

3. There's an identity element, our point at infinity $O$. For all points on the curve $P$, $P+O = O+P = P$.

4. Each element has an inverse element. This is easiest explained visually TODO: Explain visually

5. The order of operations doesn't matter, $P+Q = Q+P$ for all $P, Q$ on the curve.

## B.1 The elliptic curve discrete log problem

TODO: explain fully

As with the regular discrete log problem, the elliptic curve discrete log problem doesn't actually have a formal proof that the operation is "hard" to perform: we just know that there is no publicly available algorithm to do it efficiently. It's possible, however unlikely, that someone has a magical algorithm that makes the problem easy, and that would break elliptic curve cryptography completely. It's far more likely that we will see a stream of continuous improvements, which coupled with increased computing power eventually eat away at the security of the algorithm.

C

# Side-channel attacks

## C.1 Timing attacks

### AES cache timing

`http://tau.ac.il/~tromer/papers/cache.pdf`

### Elliptic curve timing attacks

TODO: Explain why the edwards form is great?

## C.2 Power measurement attacks

TODO: Say something here.

# Part V

# Glossary

**AEAD** Authenticated Encryption with Associated Data

**AEAD mode** Class of *block cipher mode of operation* that provides authenticated encryption, as well as authenticating some unencrypted associated data

**AES** Advanced Encryption Standard

**AKE** authenticated key exchange

**ARX** add, rotate, XOR

**asymmetric-key algorithm** See *public-key algorithm*

**asymmetric-key encryption** See *public-key encryption*

**BEAST** Browser Exploit Against SSL/TLS

**block cipher** Symmetric encryption algorithm that encrypts and decrypts blocks of fixed size

**Carter-Wegman MAC** Reusable *message authentication code* scheme built from a *one-time MAC*. Combines benefits of performance and ease of use

**CBC** cipher block chaining

**CBC mode** Cipher block chaining mode; common mode of operation where the previous ciphertext block is XORed with the plaintext block during encryption. Takes an initialization vector, which assumes the role of the “block before the first block”

**CDN** content distribution network

**cross-site request forgery** Kind of attack where a malicious website tricks the browser into making requests to another website. Can be prevented by properly authenticating requests instead of relying on ambient authority such as session cookies

**CSPRNG** cryptographically secure pseudorandom number generator

**CSRF** *cross-site request forgery*

**CTR mode** Counter mode; a *nonce* combined with a counter produces a sequence of inputs to the block cipher; the resulting ciphertext blocks are the keystream

**DES** Data Encryption Standard

**ECB mode** Electronic code book mode; mode of operation where plaintext is separated into blocks that are encrypted separately under the same key. The default mode in many cryptographic libraries, despite many security issues

**encryption oracle** An *oracle* that will encrypt some data

**FIPS** Federal Information Processing Standards

**GCM** Galois Counter Mode

**GCM mode** Galois counter mode; *AEAD mode* combining *CTR mode* with a *Carter-Wegman MAC*

**GMAC** *message authentication code* part of *GCM mode* used separately

**HKDF** HMAC-based (Extract-and-Expand) Key Derivation Function

**HMAC** Hash-based Message Authentication Code

**HSTS** HTTP Strict Transport Security

**initialization vector** Data used to initialize some algorithms such as *CBC mode*. Generally not required to be secret, but required to be unpredictable. Compare *nonce*, *salt*

**IV** *initialization vector*

**KDF** key derivation function

**key agreement** See *key exchange*

**key exchange** The process of exchanging keys across an insecure medium using a particular cryptographic protocol. Typically designed to be secure against eavesdroppers. Also known as key agreement

**keyspace** The set of all possible keys

**MAC** message authentication code

**message authentication code** Small piece of information used to verify authenticity and integrity of a message. Often called a tag

**MITM** man-in-the-middle

**mode of operation**

**modes of operation** Generic construction that encrypts and decrypts streams, built from a block cipher

**nonce** **N**umber used **once**. Used in many cryptographic protocols. Generally does not have to be secret or unpredictable, but does have to be unique. Compare *initialization vector*, *salt*

**OCB** offset codebook

**OCB mode** Offset codebook mode; high-performance *AEAD mode*, unfortunately encumbered by patents

**one-time MAC** *message authentication code* that can only be used securely for a single message. Main benefit is increased performance over re-usable *MAC*

**oracle** A “black box” that will perform some computation for you

**OTR** off-the-record

**OTR messaging** Off-the-record messaging, messaging protocol that intends to mimic the properties of a real-life private conversation. Piggy-backs onto existing instant messaging protocols

**PRF** pseudorandom function

**PRNG** pseudorandom number generator

**PRP** pseudorandom permutation

**public-key algorithm** Algorithm that uses a pair of two related but distinct keys. Also known as *asymmetric-key algorithm*. Examples include *public-key encryption* and most *key exchange* protocols

**public-key encryption** Encryption using a pair of distinct keys for encryption and decryption. Also known as asymmetric-key encryption. Contrast with *secret-key encryption*

**RSA** Rivest Shamir Adleman

**salt** Random data that is added to a cryptographic primitive (usually a one-way function such as a cryptographic hash function or a key derivation function) Customizes such functions to produce different outputs (provided the salt is different). Can be used to prevent e.g. dictionary attacks. Typically does not have to be secret, but secrecy may improve security properties of the system. Compare *nonce*, *initialization vector*

**secret-key encryption** Encryption that uses the same key for both encryption and decryption. Also known as symmetric-key encryption. Contrast with *public-key encryption*

**SMP** socialist millionaire protocol

**stream cipher** Symmetric encryption algorithm that encrypts streams of arbitrary size

**substitution-permutation network** Generic design for block ciphers where the block is enciphered by repeated substitutions and permutations

**symmetric-key encryption** See *secret-key encryption*

# Index

# Part VI

# References

# Bibliography

[fip01] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[gcm07] NIST special publication 800-38d: recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. November 2007. URL: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[ABP+] Nadhem AlFardan, Dan Bernstein, Kenny Paterson, Bertram Poettering, and Jacob Schuldt. On the security of RC4 in TLS and WPA. URL: http://www.isg.rhul.ac.uk/tls/.

[AV96] Ross Anderson and Serge Vaudenay. Minding your p's and q's. In *In Advances in Cryptology - ASIACRYPT'96, LNCS 1163*, 26–35. Springer-Verlag, 1996. URL: http://www.cl.cam.ac.uk/~rja14/Papers/psandqs.pdf.

[BK12] Elaine Barker and John Kelsey. Nist special publication 800-90a recommendation for random number generation using deterministic random bit generators. 2012. URL:

http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf.

[Bel06] Mihir Bellare. New proofs for NMAC and HMAC: security without collision-resistance. 2006. URL: http://cseweb.ucsd.edu/~mihir/papers/hmac-new.html.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In 1–15. Springer-Verlag, 1996. URL: http://www.ssrc.ucsc.edu/PaperArchive/bellare-lncs96.pdf.

[BN07] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: relations among notions and analysis of the generic composition paradigm. 2007. URL: http://cseweb.ucsd.edu/~mihir/papers/oem.pdf.

[BR95] Mihir Bellare and Phillip Rogaway. Optimal Asymmetric Encryption – How to encrypt with RSA. *Advances in Cryptology - EUROCRYPT '94 - Lecture Notes in Computer Science*, 1995. URL: http://www-cse.ucsd.edu/users/mihir/papers/oae.pdf.

[Ber] D. J. Bernstein. Snuffle 2005: the Salsa20 encryption function. URL: http://cr.yp.to/snuffle.html#speed.

[BDK+09] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key recovery attacks of practical complexity on AES variants with up to 10 rounds. Cryptology ePrint Archive, Report 2009/374, 2009. URL: http://eprint.iacr.org/2009/374.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256.

Cryptology ePrint Archive, Report 2009/317, 2009. URL: `http://eprint.iacr.org/2009/317`.

[BHK+] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. RFC 4418: UMAC: Message Authentication Code using Universal Hashing. URL: `https://www.ietf.org/rfc/rfc4418.txt`.

[BHK+99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and Secure Message Authentication. 1999. URL: `http://www.cs.ucdavis.edu/~rogaway/papers/umac-full.pdf`.

[Bon99] Dan Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46:203–213, 1999. URL: `http://crypto.stanford.edu/dabo/papers/RSA-survey.pdf`.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, 2004. URL: `https://otr.cypherpunks.ca/otr-wpes.pdf`.

[BGjosteen07] Daniel R. L. Brown and Kristian Gjøsteen. A security analysis of the nist sp 800-90 elliptic curve random number generator. Cryptology ePrint Archive, Report 2007/048, 2007. URL: `http://eprint.iacr.org/2007/048.pdf`.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2.

[Dai] Wei Dai. Crypto++ 5.6.0 benchmarks. URL: `http://www.cryptopp.com/benchmarks.html`.

[dBB93] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of MD5. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT 1993*, volume 765 of Lecture Notes in Computer Science, 293–304. Lofthus,N, 1993. URL: https://www.cosic.esat.kuleuven.be/publications/article-143.pdf.

[DR] T. Dierks and E. Rescorla. RFC 5246: the transport layer security (TLS) protocol, version 1.2. URL: https://tools.ietf.org/html/rfc5246.

[ECR] ECRYPT. Measurements of SHA-3 finalists, indexed by machine. URL: https://bench.cr.yp.to/results-sha3.html.

[FS99] Niels Ferguson and Bruce Schneier. A cryptographic evaluation of ipsec. 1999. URL: https://www.schneier.com/paper-ipsec.pdf.

[FMS01] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In 1–24. 2001. URL: http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Rc4_ksa.ps.

[Gmb08] SciEngines GmbH. Break DES in less than a single day. 2008. URL: http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html.

[HJB] J. Hodges, C. Jackson, and A. Barth. RFC 6797: http strict transport security (HSTS). URL: https://tools.ietf.org/html/rfc6797.

[Hol] S. Hollenbeck. RFC 3749: transport layer security protocol compression methods. URL: https://tools.ietf.org/html/rfc3749.

[Hou] R. Housley. RFC 5652: cryptographic message syntax (CMS). URL: https://tools.ietf.org/html/rfc5652#section-6.3.

[Hua] Sinan Huang. Hardware evaluation of SHA-3 candidates. URL: https://theses.lib.vt.edu/theses/available/etd-05172011-141328/unrestricted/Huang_S_T_2011.pdf.

[JY02] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. 2002. URL: http://cr.yp.to/bib/2003/joye-ladder.pdf.

[Kle08] Andreas Klein. Attacks on the RC4 stream cipher. *Des. Codes Cryptography*, 48(3):269–286, September 2008. URL: http://cage.ugent.be/~klein/papers/RC4-en.pdf, doi:10.1007/s10623-008-9206-6.

[Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). 2001. URL: http://www.iacr.org/archive/crypto2001/21390309.pdf.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: the HKDF scheme. Cryptology ePrint Archive, Report 2010/264, 2010. URL: http://eprint.iacr.org/2010/264.

[KE] Hugo Krawczyk and Pasi Eronen. RFC 5869: HMAC-based extract-and-expand key derivation function (HKDF). URL: https://tools.ietf.org/html/rfc5869.

[Lab] RSA Laboratories. What key size should be used? URL: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/key-size.htm.

[LWdW05] Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding x.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005. URL: http://eprint.iacr.org/2005/067.

[Mar11] Moxie Marlinspike. The cryptographic doom principle. 2011. URL: http://www.thoughtcrime.org/blog/the-cryptographic-doom-principle/.

[MWES06] Joshua Mason, Kathryn Watkins, Jason Eisner, and Adam Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In *Proceedings of the 13th ACM conference on Computer and Communications Security*, CCS '06, 235–244. New York, NY, USA, 2006. ACM. URL: http://www.cs.jhu.edu/~jason/papers/mason+al.ccs06.pdf, doi:10.1145/1180405.1180435.

[MHMP13] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. Cryptology ePrint Archive, Report 2013/346, 2013. URL: http://eprint.iacr.org/2013/346.pdf.

[NS00] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Digital Signature Algorithm with partially known nonces. *Journal of Cryptology*, 15:151–176, 2000. URL: ftp://ftp.ens.fr/pub/dmi/users/pnguyen/PubDSA.ps.gz.

[Rog] Philip Rogaway. OCB - An Authenticated-Encryption Scheme - Licensing. URL: http://www.cs.ucdavis.edu/~rogaway/ocb/license.htm.

[SS08] Somitra Kumar Sanadhya and Palash Sarkar. New collision attacks against up to 24-step SHA-2. 2008. URL: http://eprint.iacr.org/2008/270.

[SS06] Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. 2006. URL: http://www.cosic.esat.kuleuven.be/wissec2006/papers/21.pdf.

[SBK+] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. URL: https://shattered.it/static/shattered.pdf.

[SKP15] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015. URL: http://eprint.iacr.org/2015/967.

[TP] S. Turner and T. Polk. RFC 6176: prohibiting secure sockets layer (SSL) version 2.0. URL: https://tools.ietf.org/html/rfc6176.

[Vau] Serge Vaudenay. Security flaws induced by CBC padding applications to SSL, IPSec, WTLS... URL: http://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf.

[WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. URL: http://eprint.iacr.org/2004/199.

[WYW+09] Xiaoyun Wang, Hongbo Yu, Wei Wang, Haina Zhang, and Tao Zhan. Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual*

*International Conference on the Theory and Applications of Cryptographic Techniques*, volume 5479 of Lecture Notes in Computer Science, 121–133. 2009. URL: http://www.iacr.org/archive/eurocrypt2009/54790122/54790122.pdf, doi:10.1007/978-3-642-01001-9_7.

[InstitutefStandardsTechnology] National Institute for Standards and Technology. Sp800-57: recommendation for key management – part 1: general (revised). URL: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf.