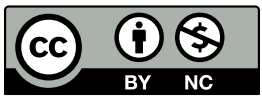
Crypto 101

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.

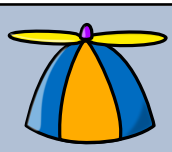
For a long time, cryptography has been deemed the exclusive realm of experts. From the many internal leaks we've seen over the years of the internals of both large and small corporations alike, it has become obvious that that approach is doing more harm than good. We can no longer afford to keep the 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

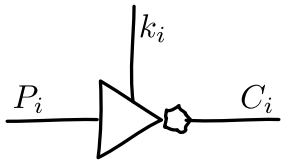
5

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.

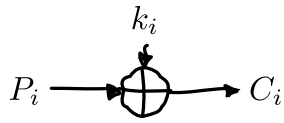
Another way to think of XOR is as something called a "programmable inverter": one input bit decides whether to invert the other input bit, or to just pass it through unchanged. "Inverting" bits is colloquially called "flipping" bits, a term we'll use often throughout the book.



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:

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

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.

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

5.4 One-time pads

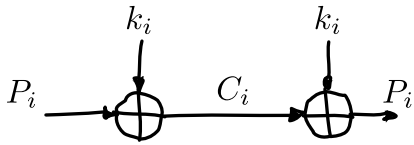
XOR may seem like an awfully simple, even trivial operator. Even so, there's an encryption scheme, called a one-time

[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]

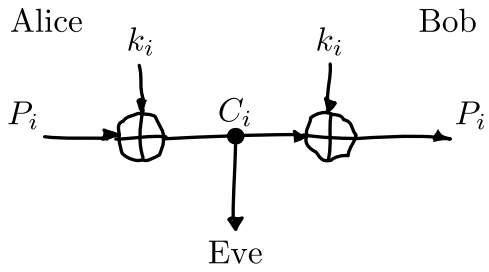
Suppose we can translate our plaintext into a sequence of bits. We also have the pad of random bits, shared between the sender and the (one or more) recipients. We can compute the ciphertext by taking the bitwise XOR of the two sequences of bits.



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

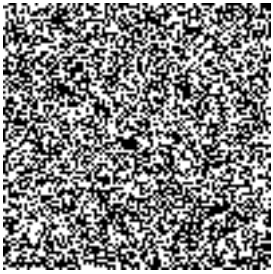
A classical approach to breaking multi-time pad systems involves "crib-dragging", a process that uses small sequences that are expected to occur with high probability. Those sequences are called "cribs". The name crib-dragging originated from the fact that these small "cribs" are dragged from left to 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

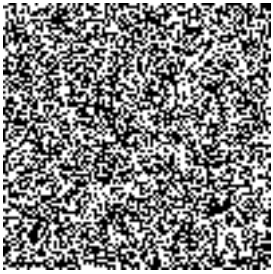
[6] We use capital letters when referring to an entire message, as opposed to just bits of a message.

(a) *First plaintext.*

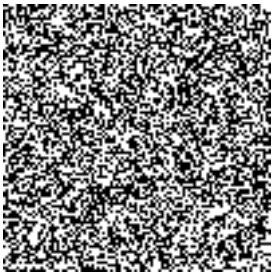
(b) *Second plaintext.*



(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.