Non-math guide to cryptography

This is an informal, non-academic, math-free discussion of cryptography ("crypto") for users, programmers, and hackers. The focus is on how it's used in the real-world, especially within web-browsers and SSL. The reader is expected to be somewhat technical, comfortable with running tools from the command-line. We'll be using the *openssl*, *pgp*, and *ssh* command-line tools in this text. For programmers, there is some source code accompanying this text.

Key points:

- There are four basic building blocks of cryptography: encryption, hashing, public-key, and randomness. Each of these has sub categories.
- The design of crypto is driven by how it's attacked, such as brute-force and man-in-the-middle.
- The design of crypto is also driven by basic principles we've worked out over the years, such as Kerckhoff's Principle.
- While it seems abstract, something only the military needs, crypto is now an essential feature of all technology we use.

The advantage, and also disadvantage, of this text is the informal discussion. This can be helpful because when you failed to "get" the concepts with one explanation, then an alternate point of view may help make things finally "click". However, if you are trying to pass a test based on conventional definitions, requiring you to phrase things conventionally, then you might miss them using this text. If you are studying for a test on crypto, many details will be missing in this text.

1   History of crypto

Encrypted messages go back to the ancient Greeks and Romans, and perhaps further. Complexity was limited by what could be done by hand, such as consulting tables to transform each letter of a message into another letter.

Around the World Wars, electro-mechanical devices like "Enigma" were used to encrypt and decrypt messages with far more complexity than what could be done by hand. The Enigma had multiple wheels to thoroughly transform a message. Using the device was as simple as configuring the password/key, then typing the message on a typewriter keyboard.

The modern era of cryptography starts around the 1970s with computers. Instead of text messages in the previous eras, mathematical algorithms transform binary data. These algorithms are as complex as they need to be in order to be essentially unbreakable.

With computers, we also find there are more uses for crypto than simply encryption. For example, **Bitcoin** is a "cryptocurrency", but nowhere does it encrypt anything.

Until the late 1990s, crypto was thought of as something only the military and intelligence agencies needed. Crypto was a "munition" tightly controlled by the government. But with the advent of the Internet, it soon became apparently that crypto was necessary for everything else, from ecommerce and online banking, to frivolous activities like posting pictures of your puppy. In the aftermath of Edward Snowden's revelations of domestic spying by the NSA, we've come to realize that even internal networks need protections from eavesdropping, and even simple metadata needs protection.

2   Magical cryptos and where to find them

This guide is not for cryptographers, so doesn't focus on math. Instead, it's for users of cryptography. This section contains a list of where you use cryptography in your daily life.

First and foremost is **SSL**, or "secure sockets layer". Most web pages you access these days start with "https://" (or a lock icon) meaning that the connection between you and the website has been encrypted with SSL. Much of the theory portion of this text is guided by the later section describing how SSL works. In particular, we'll be exploring why and how those annoying certificate errors happen. Not that SSL doesn't mean the site is secure or trustworthy, only that your connection to the site is secure from eavesdropping.
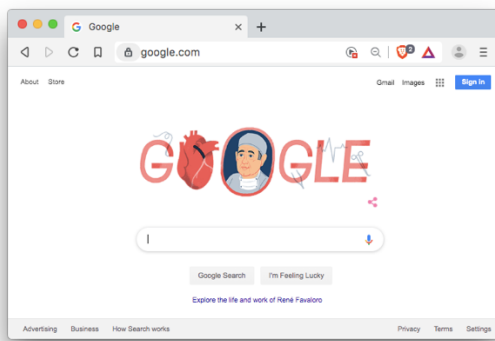

Figure 2-1Screenshot of webbrowser, by Robert Graham

Most people have mobile phones these days. They use something called **full disk encryption**, meaning that everything (essentially) on the device is encrypted. If the phone is lost or stolen, the stranger will not be able to read anything on the device without unlocking it.

This encryption works with **hardware enclaves** within the device. A typical 6-digit passcode in an iPhone has only a million combinations, which sounds like a lot if you trying to type them in, but a computer reading the disk can test millions per second. By including the secret code in a separate chip, instead of on the drive, that chip can force the hacker to slow down, and then discard the keys (thus wiping the data) after 10 failed attempts.

Such hardware is found in other places, such as your **credit card**. In the past, your card really had no security. The information on the magnetic stripe on the back just contained the same information as printed on the front. The chip, however, contains secret information available nowhere else. The chip itself verifies the transaction using crypto, rather than giving that information to the chip reader.

Key fobs, which are popular for locking car doors or opening garage doors, are likewise small cryptographic devices. They use enough cryptography to prevent evil doers from guessing the codes they send, but also, to prevent them from capturing the radio signals and replaying them.

Much online communication, such as using Apple's FaceTime for video phone calls, or Facebook's WhatsApp to send messages, is based upon **end-to-end encryption**. That means the even the company providing the service cannot eavesdrop on the communications. This doesn't protect against a virus infecting the phone from intercepting things, but does stop evil corporate advertisers or autocratic government regimes.

Emails are still sent unencrypted, mostly. You do have the option of sending encrypted emails with **S/MIME** or **PGP**, though this is more trouble than most people are willing to put up with. However, even though they aren't encrypted, most emails these days are signed with something called **DKIM**. You probably don't notice it, but it's key part of how your email provider identifies and blocks spam and phishing attempts.

Video and music is encrypted with **DRM** to prevent copying. This protects the content, as stored on blueray discs or streamed on satellite/cable premium channels, but it also protects the signal going across the HDMI cables between a device/computer and the TV/monitor. Devices and computers all have secret keys in them to prevent content from being digitally copied.

Many employees now have **employee badges**. They have to wear these inside the building so that guards can physically inspect them, matching the name with the persons photograph. They are also used to "swipe" across a reader to open locked doors, either to get into the building itself, or special rooms that only some employees are authorized to enter. The readers bounce radio signals off the badge to identify them ("RFID"). In theory, crypto is used to protect this quick communication, but in practice, many such badges can be cloned by a hacker standing next to you with a cloner in their pocket. About 1% of the population uses the DoD "CAC" card, which includes a smart chip using X.509 certificates to verify a person's identity.

If not part of the employee badge, many have a separate **two-factor authentication** (2FA) key fob. These are either battery powered devices displaying a sequence of numbers, a device plugged into the USB port, or an RFID device.

Instead of remembering individual passwords for all websites, you might use a **password manager** instead. These are services which create a unique, and strong, password for every website you visit, with you only having to memorize a single password for that service.

Your computer/phone regularly downloads and applies **software updates**. These updates are usually cryptographically signed by the maker, so that hackers can't intercept and add malicious code to them.

These days, instead of creating accounts for every website you visit, they'll let you login via your social media accounts (like Facebook or Twitter). This uses a cryptographic technology called **OAth**, allowing third party websites to authenticate you without discovering your secret information.

You probably use **WiFi** and **Bluetooth**. These radio protocols wouldn't work without authentication and encryption. Sure, public WiFi without encryption is used, but that only works because all sensitive websites use SSL.

3　Encryption

There are four building blocks to cryptography: **encryption**, hashing, public-keys, and random numbers. Encryption is the part of cryptography you are most familiar with; you might've thought encryption was the *only* part of cryptography.

Some key points:

- Encryption uses a secret **key** to convert a **message** into what looks like random garbage so that anybody intercepting it cannot read the contents.
- This key is a **shared secret**, known both to the sender and receiver of the message, but nobody else.
- The most popular **algorithm** used today is known as AES.
- The **key size** is the biggest factor that determines the strength against hacking.
- A **nonce** or **initialization vector** must be used to prevent the same data, when encrypted a second time, from being encrypted the same way.
- Encryption must also be checked against tampering (**authenticated encryption)**, though that's discussed in the next section on *Hashing*.

3.1　Encryption introduction

An example of early encryption was the one used by the Knights Templar[1] [2] [3], the Catholic military order that was responsible for European banking from around 1100 AD to 1300 AD. They used a simple **substitution cipher** to transform letters into arcane symbols. They used an image like the following to lookup each letter and substitute it with a symbol.
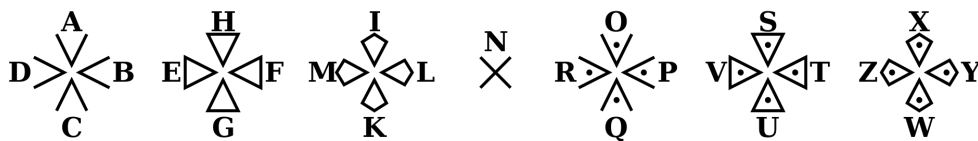

Figure 3-1Knights Templar cipher, public-domain image from Wikipedia

Hypothetically, let's say that you want to use this cipher to send a secret message to your compatriots "Attack at midnight". You'd lookup each letter by hand and draw the corresponding symbol, creating a message that looks like the following[4]:

Figure 3-2 Ciphertext of "Attack at midnight"

---

[1] See also *The Da Vinci Code* by Dan Brown [fiction].

[2] See also *Foucault's Pendulum* by Umberto Eco [fiction].

[3] See also *Ivanhoe* by Sir Walter Scott [fiction].

[4] Online Knights Templar cipher https://www.dcode.fr/templars-cipher

These symbols are chosen this way to make *decryption* easy, so that given the above image, we can quickly find the symbol and corresponding letter to translate back again.

Instead of arcane symbols, we could just substitute letters for other letters. For example, we might use a table that looks like the following:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
JHMZQTIEDSVWGRCPYFABKNLOXU
```

In this example, we'll replace the letter 'A' with a 'J', the letter 'B' with an 'H', and so on. Doing this with our message "Attack at midnight", we'll create a message that looks something like the following:

```
JBBJMV JB GDZRDIEB
```

The receiver of this message would then use a decryption table like the following to reverse the translation:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
STOIHRMBGAUWCVXPENJFZKLYQD
```

Instead of a substation table, we could instead use math to transform them. The Roman's used an encryption algorithm that today we call the **Caesar Cipher**. This *algorithm* consists of shifting the letters of the message by a certain number of places in the alphabet.

Let's use the Caesar Cipher to encrypt the message "Attack at midnight", shifting each letter by 3 places in the alphabet:

- A becomes D
- B becomes E
- C becomes F
- D becomes G
- ….
- V becomes Y
- W becomes Z
- X becomes A (wraps around)
- Y becomes B (wraps around)
- Z becomes C (wraps around)

The resulting encrypted message becomes:

```
DWWDFN DW PLGQLJKW
```

Decryption using the Caesar cipher works in reverse, shifting all the letters the other direction.

3.2   Plain-text, cipher-text, and messages

Let's introduce some technical terms here. In the above examples, we call the original message the **plain-text** (or **plaintext**, without the hyphen). We call the encrypted version the **cipher-text** (or **ciphertext**).

So far we've been using the term **message**, because that's what was sent historically, short messages sent between people. But in the modern age of computers, we mean so much more by this. By *message* we can mean any sort of **data**, such as **files**, the entire **disk**, **emails**, **instant messages** on the phone, streams of **network** data, and so forth.

The above examples use human-readable **text**, as this is what the ancients encrypted. But with modern computers, we work with **binary** data instead. A modern *message* is therefore understood to be a series of bits. It's a lot harder showing such messages as examples, because your eyes will soon glaze over trying to understand a long sequence of bits, such as the following:

```
1110000011010011110011110110011001011111
1010110101110000000100101110010111101011
```

For computer data, we don't show the binary, but use a more compact form known as **hexadecimal** or **hex**. Four binary digits have 16 possible combinations, so we represent every four bits with a symbol of either the ten digits (0..9) or one of the first six letters of the alphabet (A..F). Thus, raw computer data often looks like:

```
4049351901d2698c994ce4e22fb336b2
```

Hexadecimal is described in more detail in the section on *Encoding* below, where we discuss yet other forms, such as BASE64. Whether straight binary, hexadecimal, or BASE64, these all refer to raw internal computer data.

3.3   Algorithms

In the introductory example, we used simple **algorithms** to encrypt data, those known to the Romans and Knights Templar. The algorithms used by modern computers are more complex. However, this is simply due to there being *more* steps in the process, rather than there being weird steps. In other words, a Knight Templar would conceptually understand the steps involved, even if there were more than they could easily do by hand.

Such algorithms are extremely difficult to design. Cryptographers are constantly attacking them, looking for weaknesses they can exploit to crack/break encrypted messages. The ones we rely upon today are those algorithms that survive the test of time, that have survived decades of the world's best cryptographers pounding upon them.

The most common algorithm used today is known as **AES** or the **Advanced Encryption Standard**. It was standardized by the US government in 2001. It's built into most hardware today, so is nearly "free", both very fast and using very little battery power on mobile devices. It's the only algorithm certified by the NSA for encrypting top secret data.

AES was needed because the previous government standard, **DES** or **Data Encryption Standard** from 1977, was obsolete, easily cracked, and hence insecure. Your neighbor's teenager can crack messages encrypted with DES. **Triple-DES**, which encrypts with DES three times, is still secure – but is about 100 times slower than AES on modern computers.[5]

There are other algorithms you might see. During the height of the dot-com era in the late 1990s, **RC4** was the most popular encryption algorithm for SSL, though like DES it is now insecure and obsolete. The **ChaCha20** algorithm is a popular alternative to AES in today's SSL. It is very fast, with software[6] implementations roughly only half as fast as hardware accelerated AES.

3.4   Keys

Algorithms are public. They contain no secrets that will protect your data. Instead, the only secret is the **key**, shared between the sender and receiver of an encrypted message.

The key is a just a binary number. Normally, we humans don't use the key directly. We don't remember a random jumble of bits. Instead, the key is either chosen at random (as in SSL), or is derived from a passcode, passphrase, or password. When you lock your phone or encrypt a ZIP file, you choose a passcode/password, and it's transformed into key using a **key derivation function**, which we'll discuss later [page 22].

It's worth repeating that your security/secrecy depends *solely* on the key. In past eras, wrong thinking people tried also to keep the algorithm secret, under the belief that the more things kept secret, the more secure things would be. Over the last 200 years, we've found the opposite to be true. The more transparent we are about the workings of crypto, *except* the secret key, the more secure we are. This is known as *Kerckhoff's Principle* [page 46]. In addition, you don't have to keep the encrypted message secret, but can post it in the NYTimes for the world to see, relying upon the fact that nobody without the secret key can ever decrypt it.

3.5   Hands on with encryption

Now that we've discussed *messages*, *algorithms*, and *keys,* let's use these concepts to encrypt a message. To do this, we are going to use the command-line tool known as *openssl* (part of the *OpenSSL* package).

---

[5] According my tests using the *openssl* command-line benchmarks, AES with typical hardware acceleration is about 100 times faster than Triple-DES.

[6] An implementation of this algorithm is in the file *util-chacha20.c* in the source code that accompanies this text, though it's written for clarity rather than performance, so is slower than optimized versions found in crypto libraries.

Let's say that we want to encrypt the message "Attack at dawn", that we want to use the algorithm "AES", and a key of "1234". The command would look like the following:

```
echo -n "Attack at dawn" | openssl enc -aes128 -K 1234 -iv 0 -out
ciphertext
```
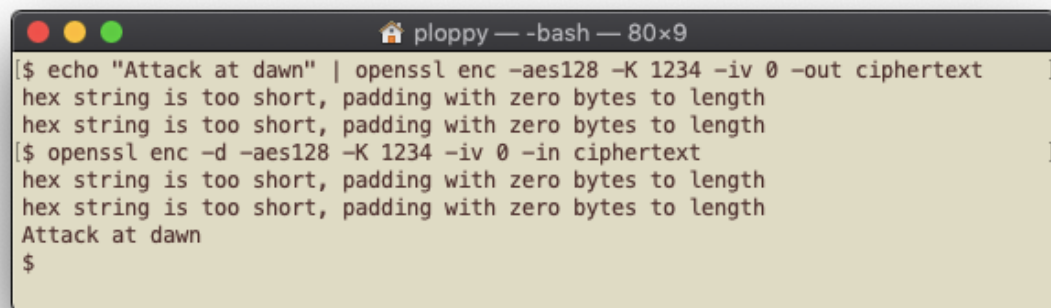
This stores the message in a file named *ciphertext* that we can send to our compatriots that will have the contents:

```
4049351901d2698c994ce4e22fb336b2
```

Our compatriots then decrypt the message by typing:

```
openssl enc -d -aes128 -K 1234 -iv 0 -in ciphertext
```

This adds the flag "-d" to mean "decrypt", and changes the "-out" flag to "-in".

```
● ● ●                    🏠 ploppy — -bash — 80×9
[$ echo "Attack at dawn" | openssl enc –aes128 –K 1234 –iv 0 –out ciphertext    ]
hex string is too short, padding with zero bytes to length
hex string is too short, padding with zero bytes to length
[$ openssl enc –d –aes128 –K 1234 –iv 0 –in ciphertext                          ]
hex string is too short, padding with zero bytes to length
hex string is too short, padding with zero bytes to length
Attack at dawn
$
```

3.6   Security strength

The difficulty in breaking the encryption is called its **security strength**. This comes partly from the algorithm strength, but mostly from the **key length**.

The older encryption standard DES (from 1977) had a short key of only 56-bits. The new encryption standard AES is stronger with a key of 128-bits. AES optionally allows even stronger keys of 256-bits.

The reason key length matters is because one technique of breaking encryption is to try all combinations of the key until you find the one that decrypts the message. Failed attempts look like a bunch of random bits, the successful decryption will result in bits that are clearly non-random.[7]

---

[7] Unless, of course, you encrypt random bits. But almost always, what we are encrypting is easily identifiable, and thus, easy for the attacker to discover when they've succeeded.

This technique is known as **brute-force**. There's nothing elegant about this. If an encryption algorithm uses a 40-bit key, then it takes 2^{40} guesses in order to test all combinations of the key, or roughly a trillion guesses (a million million).

The most important concept in brute forcing is **exponential difficulty**. An 80-bit key is not twice as difficult to brute-force as a 40-bit key, but a *trillion* times more difficult. A key twice as difficult as a 40-bit key would be 41-bit key – each additional bit doubles the difficulty.

A 40-bit key can easily be cracked by a desktop computer in about a day, whereas the trillion-times-more-difficult 80-bit key is beyond the abilities of even the NSA. The current standard, 128-bit crypto, is 256 trillion times more difficult than that.

Let's graph this. Let's look at three options available to us, cracking these keys with the *Raspberry Pi 4*, a $35 hobbyist computer, a $1000 desktop computer with a graphics processor (GPU) optimized for number crunching, and the NSA who buys a *million* such computers. Let's graph the time it takes for each option to crack keys, as keys get longer.
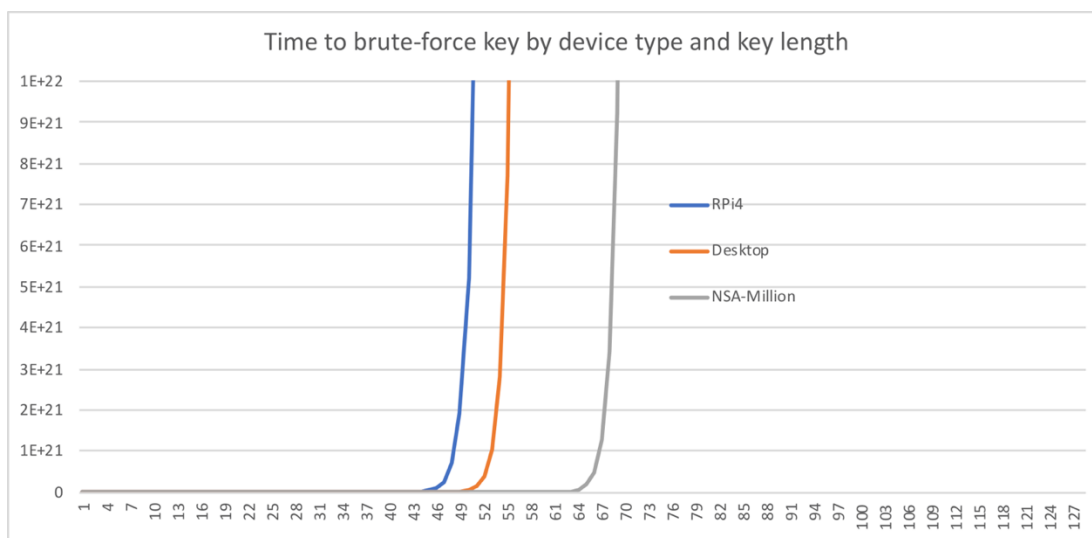
The first thing to notice in this graph is how quickly exponential growth makes the amount of time explode. Short keys can be cracked almost instantly, long keys take forever, and there is only a sliver between them.

The second thing to notice is the difference in capabilities between a $35 computer and a $1-billion. Keys shorter than 48-bits can be cracked by everyone, including your neighbor's teenager using money they earned babysitting. Keys longer than 68-bits can't be cracked by anyone, not even the NSA. Today's keys are 128-bits long – solidly in the "uncrackable" territory.

This shows why DES, the 1977 encryption standard, is obsolete. It used 56-bits, right within that territory where it can be cracked. This also shows why even with Moore's Law regularly doubling computer speed every 18 months why it will take another century before 128-bit keys can be cracked.

This also shows why the TV/movie cliché of hackers just trying a little harder to "bypass the encryptions" doesn't reflect reality. They can either do it trivially with a Raspberry Pi because the encryption is irrelevant, or with relevant encryption using 128-bits, it's nowhere in the realm of possibility, even if they wrote a hack that somehow took over all the world's computers to work on the task.

3.7   Stream-ciphers and the one-time-pad

There are two types of encryption algorithms, the *stream-cipher* and the *block-cipher*. In this subsection we are going to discuss the **stream-cipher**.

The discussion starts with an old encryption algorithm known as the **one-time-pad**. Instead of using a short key, it uses a pad of random data (numbers or letters). The length of this "key" is the same length as the message that needs to be encrypted. To encrypt a message, you'd rip out the next page in the booklet (or pad of paper), and **mix** each letter/number on that page with the corresponding letter in your message.

There are various ways we can mix the pad with the message. One way is like the Substitution Cipher described in the simple examples above, with a two-dimensional lookup table. Another way is like the Caesar Cipher, which uses a number to shift the letter in the alphabet. This shift method is what we use in our example below.

In our hypothetical example, we are going to use a page that contains the following numbers, and then mix those numbers with the letters of our message:

```
24 15 11 16 23  1  8 13
18  9 18  4 12 20 22 10
. . .
```

In order to encrypt a message, like "Attack at midnight", you'd line them up, and shift each letter in the alphabet the corresponding number of times. In other words, if the input letter is 'A', and you shift by '2', you get the new letter 'C'. If you go past 'Z', just wrap back around and start from 'A'.

Thus, to encrypt our message with our one-time-pad, we get:

```
 A  t  t  a  c  k  a  t  m  i  d  n  i  g  h  t
24 15 11 16 23  1  8 13 18  9 18  4 12 20 22 10
 Y  i  e  q  z  l  i  g  e  r  v  r  u  a  d  d
```

A one-time-pad cannot be cracked. In our simple examples [page 5], the same letter would be encrypted the same way every time. In the one-time-pad, each letter will be encrypted differently. Thus, the letter 'A' in the plaintext gets encrypted variously as 'Y', 'Q', 'I' in the ciphertext. Conversely, we see the letter 'd' twice in the ciphertext at the end, but each time it represents different letters.

We can prove, with math, that this means a one-time-pad can't be cracked, as long as the key is as long as the message, the key is perfectly random, and the key is never reused. We still have such World War era messages captured from spies that we've never been able to decrypt.

While one-time-pads are perfectly secure, they are also usually worthless. People aren't going to carry around large pads with them in order to encrypt small messages. Moreover, it means the enemy has a good chance of capturing pads.

Instead of pads of truly random data we could instead generate this sequence of numbers with a mathematical algorithm, based on a short key. That's the basis for a **stream-cipher**. It uses a key to "seed" the algorithm, then generates a stream of **pseudo random** numbers, known as the **keystream**. As long as the key is long enough (at least 128-bits these days), and the algorithm cryptographically secure, then this encryption should be secure.

The above example of a one-time-pad uses text, but these days we use binary. Therefore, the way we combine a plaintext message with the keystream is different. We use the binary math operation called *xor* to do the combination. How *xor* works is described in a section below [page 51]. However, the fact that we use *xor* isn't so important. We could use *addition* instead, such as adding each byte of text with a byte of keystream, and it would still be the same security.

RC4, the encryption algorithm used during the dot-com era for SSL, was such a stream-cipher. These days, **ChaCha20** is a popular stream-cipher. This text includes accompanying code that implements this, showing how the keystream is generated, and how it's mixed with the plaintext to form the ciphertext.

To repeat: a stream-cipher doesn't encrypt the data directly. It instead generates a stream of random bits, which are then combined with the data.

3.8   Block-ciphers and stream-ciphers

As mentioned above, there are two types of encryption algorithms, the stream-cipher and the block-cipher. The way a **block-cipher** works is by encrypting data a block at a time. Unlike it a stream-cipher, it operates directly on the plaintext, instead of generating a keystream.

Crypto algorithms inherently work at the level of a "block" of data. That's because they like to shuffle things around, so that when 1 bit of input changes, this influences changes on a wide range of other bits in the block. Things that start out near the start of the block gets moved around to the end, or the middle. Any bit that changes in the key or the plaintext block should change all the bits in the ciphertext block.

AES has a block-size of 128-bits. That means everything it encrypts is done at a multiple of 128-bits. Usually we need to pad a message to fit a full block. In the example above of encrypting "Attack at dawn", we only have 112-bits of plaintext. It was padded out to 128-bits before it was encrypted, producing 128-bits of result. Adding a couple letters to the message would result in two blocks being encrypted, or 256-bits total.

The problem with block-ciphers is that, for a given key, every plaintext block with the same content will encrypt into the same ciphertext block. In other words, the text "Attack at dawn" appearing in one block will be encrypted the same if it appears the same way in a later block.

This can be a problem, as demonstrated by the famous **ECB Penguin**, where an image of a penguin is encrypted. Because much of the image is composed of identical

blocks of all-black pixels or all-white pixels, even when encrypted, we can still make out the rough image of a penguin. This is shown below:
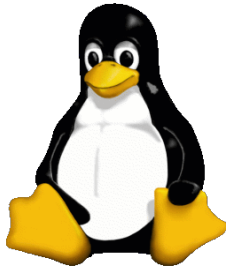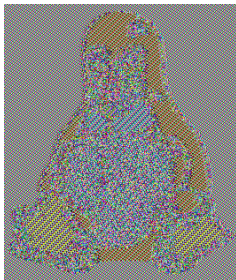

Figure 3-3 Tux, the Linux mascot

Figure 3-4 Tux encrypted with ECB mode

There are a couple simple solutions to this problem. One solution is known as **chaining mode**, where the output of a previous block is included as input into encrypting the next block. This means every block will be encrypted differently from every other block, even when it has the same contents. With chaining, the above image is now encrypted as the following image:

Figure 3-5 Tux encrypted with CBC mode

Another way of dealing with the problem is known as **counter mode**. In this mode, instead of encrypting the block of data, the algorithm is used to encrypt a counter (0, 1, 2, 3, 4, …) to produce a keystream, which is then mixed with the plaintext to get the ciphertext.

In other words, a block-cipher in counter-mode is the same as a stream-cipher. Much of the real-world use of AES is actually in some sort of counter-mode, whether it's encrypting WiFi (AES-CCMP) or SSL (AES-GCM). Thus, even though the most popular encryption algorithm is a block-cipher, much of the world's data is encrypted as with a stream-cipher mode.

3.9   Nonces

The above section uses the *ECB Penguin* to demonstrate the problem when the same block of data **within a message** can be encrypted the same way, requiring chaining or counter modes to make sure each block gets encrypted differently. There is a related problem when the same key is used to encrypt **multiple messages** that start with the same data.

For example, let's encrypt the messages "Attack at 6am" and "Attack at 7am". We get the two binary messages:

```
d342375781c58de50fe50115b2
d342375781c58de50fe50015b2
```

This results in messages that are almost identical, except for the few bits that have changed from one to the other. This gives the attacker a lot of information, even if they can't completely decrypt the message.

The way to solve this is to include some additional information into the encryption process to make sure these two messages will be encrypted differently. This is variously called a **nonce**, an **initialization vector**, or a **salt**, depending on context.

The above examples were produced with the *openssl* command-line program. It requires an initialization vector, which we supplied as 0 for both messages. The first encrypted message was created with the command:

```
echo -n "Attack at 6am" | openssl enc -chacha20 -K 1234 -iv 0
```

When creating the second message, we can increment the initialization vector by one:

```
echo -n "Attack at 7am" | openssl enc -chacha20 -K 1234 -iv 1
```

This produces two messages that now look like:

```
d342375781c58de50fe50115b2
9fe1bcb03f8513c92f1d328863
```

As you can see, because of the change in the nonce/IV, the ciphertext is completely different even though the plaintext is nearly the same, and they are encrypted with the same key.

The word "nonce" comes from old English meaning something temporary, only for this one occasion. That's exactly what this is: each nonce needs to be used only once, for one message only, and never repeated in the future.

The word "initialization vector" comes from the way it's implemented, the way that inside the algorithm it's used to initialize the encryption algorithm. The word "vector" means is a sequence of multiple bits. We often abbreviate the term to simply "IV".

The word "salt" is an analogy to how adding a little bit can dramatically change the taste. This term isn't used as much as *nonce* or *IV*.

Usually, the nonce can be **public**. It doesn't need to be secret. It's perfectly fine if your adversary knows it. Again, in encryption, it's only the key we wish to keep secret.

Often, the nonce can be **predictable**, such as a sequence number going from 1, 2, 3, 4, and so on. Other times the nonce needs to be **random**. Some prominent crypto failures have happened because an algorithm required a random nonce, but the implementers gave it a predictable nonce.

At the start of this subsection, we described encryption the same way most texts do, that encryption is simply an *algorithm* and *key* to encrypt the message. In reality, this should include *nonce* as part of the minimal set of functionality.

3.10 Authentication encryption (part 1)

Encryption algorithms only transform the data, but cannot detect if an adversary has tampered with it. To show why this is a problem, we can use the example above where we encrypt a message "Attack at 6am" and get the following output:

```
d342375781c58de50fe50115b2
```

Let's say the attacker changes the ciphertext, the hex '1' near the end to a '0' to create the following::

```
d342375781c58de50fe50015b2
```

As we saw in the previous subjection, this changes the plaintext message to "Attack at 7am". Now the adversary making this change can't predict which time this will be changed to, but they can often rely upon the fact that it'll be changed to some time other than whatever was originally specified, throwing our military offensive into disarray.

To guard against this, we need to verify the **authenticity** or **integrity** of the message, that it contains exactly what the original sender intended, and not some deliberate tampering or corruption.

When we look at how AES is most often used in SSL, it's not simply AES-CTR (AES in counter mode), but AES-GCM (AES in Galois Counter Mode). GCM contains an additional algorithm to verify data hasn't been corrupted.

Likewise, SSL doesn't use raw ChaCha20, but ChaCha20 with Poly1305, an algorithm that verifies the authenticity of the data.

We started with this section describing how encryption involved an *algorithm* and a *key*. We then went on to explain how it also required a *nonce/IV*. We finish here by explaining it also needs an algorithm and additional output to verify *authenticity*.

These algorithms will be discussed in more detail below in the section on *Hashing* [page 21].

3.11 Addition encryption terms

We often refer to this sort of encryption with the longer names **secret key encryption, symmetric encryption**, or **bulk encryption** when differentiating it with the *public-key encryption* described in the section below. These all refer to the same thing described in this section.

3.12   Summary of encryption

Encryption starts with the plaintext:, a message, file, stream of network data, or any other data.

Then using a shared secret (that both the sender and receiver know), also called the *key*, the plaintext is transformed via an *algorithm* into ciphertext.

This should include a *nonce* or *initialization vector* so that multiple similar messages should encrypt into utterly different ciphertexts.

This should include an algorithm to verify the authenticity of the message, that it hasn't been tampered with.

When sending the message, we need to include the nonce at the front so that the recipient can use it to decrypt the message, then the additional few bits at the end to verify the authenticity that the message hasn't been tampered with.

4   Hashing

There are four building blocks to cryptography: encryption, **hashing**, public-keys, and random numbers. Some key points:

- A **hash** uses an algorithm to create a unique fingerprint of a message[8] to verify integrity/authenticity, that an adversary hasn't changed the data.
- A hash algorithm generates a **fixed size** fingerprint (usually 256-bits) regardless of the amount of data it's hashing, whether it's 0 bytes or a trillion bytes.
- A hash is **one way**, so that given a hash, you can never recreate the original message that produced the hash.
- A hash typically doesn't have a secret key (like *encryption* or *public-keys*), but when it does, it's known as a **keyed hash** or **message authentication**.
- There are separate algorithms for detecting changes in fixed-sized message or files, and for detecting tampering streams of network data (aka. **authenticated encryption**).

4.1   How hashing works

The purpose of hashing is to detect whenever data has been **tampered with**, either maliciously or accidentally. This is also called verifying the **authenticity** or **integrity** of the message. The fundamental principle is that the slightest change, no matter how small, creates a wholly new fingerprint.

This is demonstrated below where we hash short messages consisting of only a single letter of the alphabet. On the left is the message in quotes (we use short messages here), on the right is the 128-bit hash, represented in hexadecimal.[9]

```
"a" = 0cc175b9c0f1b6a831c399e269772661
"b" = 92eb5ffee6ae2fec3ad71c777531578f
"c" = 4a8a08f09d37b73795649038408b5f33
"d" = 8277e0910d750195b448797616e091ad
"e" = e1671797c52e15f763380b45e841ec32
"f" = 8fa14cdd754f91cc6554c9e71929cce7
```

---

[8] As usual, whenever we say "message", we also mean a document, a file on the disk, a stream of network traffic, and any other sort of data.

[9] These results were produced using the *openssl* command utility with the MD5 algorithm, such as:
```
echo -n "a" | openssl dgst -md5
```

MD5 was chosen because it produces a nice short output for demonstration purposes, but it's insecure so we'd never use that algorithm otherwise.

The size of the hash is always the same, regardless of the size of input. In the table below, we hash increasingly long messages containing only the letter "a". The resulting hash always has the same length:

```
    "a" = 0cc175b9c0f1b6a831c399e269772661
   "aa" = 4124bc0a9335c27f086f24ba207a4912
  "aaa" = 47bce5c74f589f4867dbd57e9ca9f808
 "aaaa" = 74b87337454200d4d33f80c4663dc5e5
"aaaaa" = 594f803b380a41396ed63dca39503542
```

Even with messages billions of bytes long, such a hash will still have only the same length. This applies whether it's a short message, or a 1080p "Game of Thrones"[10] episode that's several gigabytes in size.

A hash is an item's **identity**. If two items have the same hash, then they are same thing, the same size, with the same contents). The hash "e83f499dd6eb769421a27017f3e108fc7b9f60f5" refers to the final episode of the *Game of Thrones*.[11] If you want to watch it, simply copy/paste that hash into google and follow the links, you can download that episode.

A hash is **one way**. Given a hash, it's impossible to go the other direction and figure out the input starting from the hash. Instead, the only way to find the starting input is **brute-force**, attempting all possible inputs until you find one that matches.[12] In the following hashes, I've increment the final byte to produce a different hash. Inputs that will result in these hashes exist in theory, but nobody knows of those inputs. It's not within our current abilities to find those inputs.

```
"a" = 0cc175b9c0f1b6a831c399e269772661
??? = 0cc175b9c0f1b6a831c399e269772662
??? = 0cc175b9c0f1b6a831c399e269772663
```

In other words, you can think of hashes as **one way encryption**, since you can never decrypt them. A better way is to think of them as one way **compression**. A hash contains all the information that was used to create it. Even in a large terabyte file, changing a single bit completely changes the hash. Hence the information represented by that bit, and all the other bits, are somehow contained within the hash, even though the hash may be only 256-bits in size. That compressed information can't ever be uncompressed, though.

We claim that a hash is **unique**, but obviously that can't be true, as it must be possible in theory for multiple things to hash to the same value, especially things larger than the hash size.

---

[10] *Game of Thrones* was a popular TV show that in the year 2019, was the most popular item downloaded via BitTorrent, a peer-to-peer protocol which identifies downloads through their hashes.

[11] By which we mean a specific recording of that episode. Different recordings will have slightly different binary contents, and hence, completely different hashes. For example, 0f63e5835f202caf81fd1b54bc706d21727d6c62 refers to a different recording of the same episode, at a lower quality.

[12] Such brute-force is done with passwords, because they are usually smaller than the hash size.

But only in theory. Like all crypto, this number of bits is so long that it's impractical to find two things that hash to the same value, either on purpose or accident. They are not unique if you are a God, only unique if you are a mortal human.

There are fewer than 2^{70} files[13] in the world, and a SHA256 hash has 2^{256} combinations. That means you have a 1 in 2^{186} chance that a file exists in the world with the same hash as your file, but different contents (even 1 single bit different).

Bitcoin uses an enormous amount of hashing to verify transactions. It's hashing rate is currently (year 2019) around 2^{66} hashes-per-second. It would take this network longer than the age of the universe to find two different transactions that hash to the same value.

Thus, for all practical purposes, a hash is a guaranteed unique fingerprint of a message, file, or other piece of data. Even if somebody wanted to, they couldn't create two things with the same hash.

This doesn't apply, of course, if the hash algorithm is **broken**. The algorithms MD5 and SHA1 have been broken such that it's now possible to create **collisions**, two documents with the same hash. For example, the SHA1 hash 191b636f80d0c74164ec9d9b3544decdaa2b7df5 refers to two different documents.[14] That's why we no longer use MD5 or SHA1, but use SHA2 or SHA3 instead.

4.2   Hash algorithms

Just like there are many algorithms for *encryption* there are many **algorithms** for *hashing*.

The most common is SHA2 (also known as SHA256 or SHA512 or other names, depending on the variant). We use MD5 in the examples above because it produces a short output more convenient for printing in text, but in practice, if you encounter a hash these days, it's usually SHA2. In the code accompanying this text you'll find a simple implementation of SHA2.

A full list of hash algorithms you are likely to find are:

- **MD5** was created by RSA (the company) in the 1990s. It was extremely popular during the dot-com era for a wide range of uses. It's now obsolete, because of its short output length (128-bits) and discovered weaknesses in the algorithm. Because of backwards compatibility, you'll still often see it today, but it's insecure.
- **SHA1** was the first government hashing standard, based on MD5 with a longer output length (160-bits instead of 128-bits). However, algorithmic weakness have been found that make it obsolete and insecure, so it should no longer be used.
- **SHA2** is the current hashing standard. It supports many variations with different output sizes, with the most popular size being 256-bits, in which case it's known as **SHA-256**. Other variants are known as **SHA-224, SHA-384, SHA-512**. In

---

[13] A rough estimate on EMC's claim the world will have 40 zettabytes of data by the year 2020. https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

[14] PoC||GTFO issue #19, "Accepted" or "Rejected" versions.

addition, there are variants with names like **SHA-512/256**, which calculate the larger hash, but then take only the first 256 bits, ignoring the rest of the bits. On modern 64-bit processors, SHA-512 is actually faster than SHA-256, because it processes data 64-bits at a time.

- **SHA3** was standardized in 2015 as an alternative to SHA2. It was feared that SHA2 might fall to the same vulnerabilities as SHA1 and MD5, so a new standard was created. However, SHA2 is still secure, so SHA3 is little used. The major feature of SHA3 is that it uses completely different constructions than SHA2, so is unlikely to be affected by any newly discovered problems in the older algorithm. Like SHA2, it supports a range of output sizes.
- **Blake2** is an algorithm designed to be fast in software, faster than SHA2 and much faster than SHA3. However, hardware accelerators and the specialized instructions in modern CPUs often make SHA2 faster in practice.

4.3   Keyed hashes and message authentication

Normally, hashing doesn't have a key. We are simply interested in the fingerprint, where both the hash and the content are public. In some cases, though, we want to add a key. This is known as a **keyed hash**, when referring to how it works. It's also known as **message authentication**, when describing why we do this – which is to verify that a message hasn't been tempered with, to *authenticate* it came from the sender unchanged.

One way to do this is just to add the key/password to the message being hashed, either prefixing it at the start or appending it to the end.

This naïve approach works for newer algorithms like SHA3 or Blake2, but when used with older algorithms like SHA2, there are some subtle weaknesses. Therefore, in practice the key is combined in an algorithm known as an **HMAC** or **hash-based message authentication code**. We add the key to the message and hash the combination. Then we add that key to that result and create a second hash. The first step adds the key to the end, the second step prefixes the key at the beginning. This stops theoretical attacks that work by prefixing or appending malicious data.

The exact details of HMAC doesn't matter so much as the fact it fixes problems with the naïve approach. Like so much in cryptography, you should fear the naïve approach. HMACs are often used even when the additional security isn't needed, simply because cryptographers are so paranoid and conservative.

Keyed hashes can be used to verify the identity of the sender, as only they know the secret key being added to the hash. However, they are more often used to verify *integrity*, that the message hasn't been changed. If you transmit data and also the hash together, then a malicious hacker could change both the data and the hash together. Adding a secret key to the hash means you can do this without the adversary being able to change the hash.

Such message authentication is one use for keyed hashes. There are other uses as well, such as when salting password hashes. This is described in some detail later in this text [page 70].

4.4   Authenticated encryption (part 2)

We use hashes to verify the integrity of single objects, like a message or file. Encrypted network streams pose a different problem. They send many chunks of data. Each chunk needs its own verification.

We could just use the existing hash algorithms for this, but they are inappropriate. They are often inefficient, with a lot of time to initialize and finalize the hash. They produce larger hash outputs than are strictly needed. Knowing that we are doing hashing and encrypting at the same time gives us the opportunity to do them in parallel, speeding things up.

Thus, when we look at things like SSL, we see choices for **authenticated encryption** that aren't based on a hash algorithm. One example is GCM or Galois Counter Mode used with AES. Another example is Poly1305, usually pared with ChaCha20. They serve the same purpose as a hash algorithm, but are appropriate for streams of network data instead.

With network traffic, we want to authenticate not only the plaintext/ciphertext but also the surrounding protocol headers. Packets have headers (SSL, TCP, IP, and so on) that are supposed to be unencrypted and publicly visible. One way of using authenticated encryption algorithms is in a construction called **AEAD** or **authenticated encryption with additional data**. The "additional data" means the additional input to the algorithm of unencrypted data (like packet headers) along with the encrypted data.

Like with encryption algorithms and hash algorithms, we have multiple choices for authenticated encryption:

- **CCM** (Counter Mode CBC-MAC) used with AES in WiFi encryption.
- **GCM** (Galois Counter Mode) used with AES in SSL.
- **Poly1305** used primarily with ChaCha20, but also useful with AES in counter mode.

Note that *authentication* in this context can be confusing. In other contexts, we use this term to refer to logging into a web server, using a username and password to verify your identity. When used in this context, *authentication* means something slightly different. We are verifying all the data is what the other side intended to send, that none of it is something the hacker inserted or tampered with. It's verifying the data is authenticate.

4.5   Hash size and equivalent security

In much the same way we measure the security level of encryption and public-keys according to key size, we measure the strength of hashes according to the size of their output. That's why we have different lengths for the SHA2 algorithm from 224 bits to 512 bits – longer is more secure, harder to crack.

Like with cracking encryption keys, what we are measuring is **brute-force**, how many attempts it takes to create a second document with the same hash as the first one.

However, there is a subtle mathematical anomaly known as the **Birthday Paradox**. If you are at a party of 23 people, what's the chance that any two people have the same birthday? The answer is 50%. This is vastly different odds than a similar question of the

chance that somebody has the same birthday as you, which is only around 5% when there are 23 people at the party.

The same applies to hashing. There are two problems. One problem is finding a second document that'll match the first. The other problem is finding any two documents that match. The first problem is known as a **preimage attack**, creating a document with a specific hash value. The second problem is known as a **collision attack**, finding any two documents with the same hash value.

Because of the Birthday Paradox, collision attacks require twice the number of bits for protection. That's why we use SHA-256 with AES-128, because the **equivalent security** for hash functions is twice the number of bits.

Older algorithms like MD5 and SHA1 are now insecure because we can successfully do collision attacks against them. However, in terms of preimage attacks, they are still secure. That's why you can still use old SHA1 certificates in modern browsers – you can still trust the certificates generated before SHA1 collision weaknesses were created, but not certificates generated afterwards.

Where collisions become important is with SSL *certificates*. These are the documents that have been signed by a *certificate authority* that verify a website is who they claim to be. All I have to do in order to pretend to be "google.com" is create my own certificate with the same hash as Google's. This would require a preimage attack, and is impossible, even with broken/insecure algorithms like MD5 and SHA1.

However, I could instead try for collisions, creating two certificates for my website (robertgraham.com) and Google's (google.com) that have the same hash value. I keep messing with some random data in the middle until I create two documents with the same hash value. I then have a certificate authority sign the certificate for my website, which likewise (unknown to the authority) creates a signature valid for this fake Google certificate.

Such attacks have been successfully done, which is why we can no longer use MD5 and SHA1 for SSL certificates.[15] SHA-256 is used for almost all certificates these days.

4.6   Key derivation functions

As discussed under the *encryption* section, a binary *key* is used to encrypt data. However, human users often use text passwords. In order to convert a password into a key we use a **key derivation function**. The simplest key derivation function is to use a hash.

For example, when encrypting something with AES-128, we first hash the password with SHA-256, and use the first 128-bits of the result as the AES key. For example, if

---

[15] Your web browser still trusts SHA1 certificates, old certificates that were generated before SHA1 collisions became practical. However, no trusted CA will sign new certificates with SHA1, because they can now be created with collisions.

want to use "Password1234" as our password, we might use the *openssl* command-line tool like the following to generate the hash:

```
$ echo -n "Password1234" | openssl dgst -sha256
SHA256(stdin)=
a0f3285b07c26c0dcd2191447f391170d06035e8d57e31a048ba87074f3a9a15
```

There is a problem with this approach, not with the algorithms, but with the humans. The password "Password1234" isn't very strong. It looks strong, being composed of both upper and lower case, plus digits, but it's weak, because the choice of those characters isn't random.

Over the last 20 years we've seen many events where popular websites get hacked, user account information get stolen, and hackers publish the passwords chosen by users. From this we've built lists, such as the million most common passwords users have chosen in the past. The password "Password1234" is near the top of that list.

Since password cracking tools[16] can guess *billions* of passwords per second, such weak passwords get quickly hacked.

To solve this, key derivation functions do more than just a quick hash. They attempt to slow down password cracking.

One way to slow this down is known as **PBKDF2** or **Password-Base Key Derivation Function 2**. This technique works by repeating the hash thousands of times, slowing hackers from guessing billions of passwords a second to merely millions per second.

This is how the WPA2 standard works for securing your home WiFi. The designers of WPA2 know you are going to pick a weak password that your neighbor's teenager could crack in seconds. For this reason, they force you to choose at least 8 characters for a password, as they become exponentially more difficult to crack the longer they become. They then repeatedly hash it around 4 thousand times to slow down password cracking speed. Your neighbor's teenager is therefore a lot less likely to guess even the simple password you put on your WiFi.

In the old days, *Wardriving* was a common practice, driving around the neighborhood looking for WiFi networks to break into. These days, it's a lot of harder with WPA2's protections.

An alternative to PBKDF2 is to use **memory hard** hash algorithms like *scrypt* or *Argon2*. These work by forcing the algorithm to use a lot of memory as well as a lot of CPU power. Number crunching is easy to accelerate, such as with graphics processors, FGPAs, or custom chips, giving attackers an inherent advantage over defenders. Adding memory to the mix dramatically reduces the advantage attackers have, making their cracking much harder, while not adding a significant burden to defenders.

4.7   Hash chains

In much the same way you might want to chain encryption (see *CBC mode* above), you may also want to chain the results of hashes. For example, if using SHA256 to

---

[16] The two most popular password hashing tools are *Hashcat* and *John-the-Ripper*.

verify the contents of each packet on a network connection like SSL, you'll want to hash not only the current packet, but also include as input the hash from the previous packet. In this way, the hash represents not only the current packet, but all the contents of the connection up to this point.

The famous *block chain* in Bitcoin is just a straightforward *hash chain*. When hashing the latest block of transactions, the resulting hash from the previous block is included. This verifies not only the current block but the entire chain of blocks going back to the beginning.

4.8   Proof-of-work

We mention how Bitcoin uses hash chains. It's also worth mentioning how Bitcoin uses hashes for proof-of-work. You can skip this part if you don't care about Bitcoin, but can be useful in understanding the difficulty of getting two hashes to match.

Each block of transactions needs to be verified, but only once by one person. Otherwise we have the "double-spending" problem, where there are competing verified blocks with different transactions. To way to make sure only one person hashes the block is by slowing down the hashing process **a lot**. We want something like PBKDF2, but instead of slowing it down a thousand times, we slow it down by a billion billion times.

We do this with the trick of specifying that the resulting hash must begin with a lot of leading bits set to zero, around 66 zero bits at the time of this writing. For example, the latest Bitcoin block at the time of this writing has the hash:[17]

```
000000000000000000172eec941cccb4bb1053627108cd6d5b3db8c804003bf8
```

Just like how finding exact matches for hashes requires brute-force, so does finding partial matches. A **Bitcoin miner** does this by taking the latest block of transactions, adding a bit of random data, then calculating the hash. They keep repeating this process, changing that bit of random data, until they get a result with enough leading zeros.

It's a brute-force lottery. All the miners are competing with each other until one finds the golden ticket. The winner collects all the transactions fees in the block (each transaction includes a bit extra that goes to the verifier of the block).

As described above, hashes are one-way. There's no way to go backwards from a hash to the original data. Thus, the only way to win this Bitcoin lottery is by doing the work of repeatedly calculating the hashes until you find one that matches. Hence, this is called a **proof-of-work**.

4.9   Hashes vs. hash tables

In programming, there is a data structure known as **hash tables**. They allow data stored at an index to be accessed directly, without having to search for the index. Such

---

[17] This is a real value. You can copy and paste this into google and find the transaction.

hash tables, or **hash maps**, are fundamental to many programming languages, like JavaScript.

These programmatic hashes are much the same as cryptographic hashes, but simpler, faster, and smaller. The word "hash" means the same in both contexts, taking large amount of data and compressing it into a small fingerprint that is probabilistically unique. For programmatic hashes, collisions are fine, they just cause an extra step when accessing data, skipping the first item that matches and grabbing the second, or third.

However, in modern times, we've come to realize that this opens things up to attack. A hacker can often figure out how to exploit the hash function to force everything to collide to the same index. Instead of a few collisions requiring a few extra steps to get the right data, it can now require thousands or millions of extra steps.

Thus, the hacker can provide an HTML page or a PDF document that causes the program that accesses it to essentially hang, spending all its time trying to resolve collisions.

A lot of code has therefore moved away from the traditional programmatic algorithms (like *murmur* or *cityhash*) toward ones designed by cryptographers. The most popular alternative is known as *SipHash*.

SipHash is a *keyed* hash algorithm. The key should be some true random data gathered at startup of the program. It should be kept secret, not revealed to the outside world. This way, every time the program runs, it'll arrange hash tables differently, in a way that hackers can't predict.

SipHash is not really a cryptographic hash in that it can be used in place of another cryptographic hash. However, it's cryptographic from the point of view that it's unguessable by attackers.

An alternative to SipHash is *highwayhash* from Google, which they claim is faster.

4.10   Hashes vs. checksums/CRCs

Those of you familiar with network protocols know that they already come with integrity checks called **checksums** or **cyclic redundancy checks (CRCs)**, depending on how they are calculated.

These are logically the same as cryptographic hashes, but are non-cryptographic. They are designed to detect **accidental** corruption, not **intentional** corruption. Attackers can easily tamper with the data such that the checksum/CRC comes out to the same value. They can also tamper with the checksum/CRC itself in order to match.

4.11   Hashes in action

The main reason for covering hashes is for **SSL certificates**, but they are used widely all over the place.

BitTorrent, a popular peer-to-peer file transfer service, uses hashes in multiple ways. Firstly, it identifies the entire download with its hash, known as a *Magnet URL*. These look something like the following:

```
magnet:?xt=urn:btih:D540FC48EB12F2833163EED6421D449DD8F1CE1F
```

BitTorrent clients maintain directories whereby the actual torrent file can be located. The torrent file itself contains hashes for individual chunks of the completed download. A 1-gigabyte download may be divided into 1-megabyte chunks, so a torrent file will have a list of 1024 hashes. This allows BitTorrent to exchange chunks with peers, while being confident some malicious peer isn't trying to disrupt the process by supplying corrupted chunks. A corrupted chunk can be discarded without throwing away all the work of downloading the other chunks.

GitHub, the popular code version tracking system, uses hashes to tell if two files are the same or whether they have been changed. In other words, if you change the timestamp on a file (the normal way operating systems detect changes), it's still the same file as far as GitHub is concerned if it has the same hash. It doesn't ever compare the contents of two files with each other, it only calculates the hashes and compares the hashes.

Anti-virus programs use hashes as one type of *signature*. Hashes of known evil files are included every time you update your signatures. In addition to blacklisting known bad files, it'll whitelist known good files that look suspicious (and might trigger other signatures/heuristics) but which the anti-virus vendor knows to be good.

Password hashing is the standard way of storing passwords. When logging in, the system hashes your password, and compares the hash with that stored in their user account database. This is why that when you forget your password, they can reset your password, but they can't tell you what your forgotten password was. They don't know your password, they only know its hash, and since hashes are one-way, they can't derive your password from the hash.

4.12   Summary of hashing

A hash takes a potentially large amount of input and compresses it down to a small fingerprint that is unique to that input. The slightest change, no matter how insignificant, will completely change the fingerprint.

Hashing is used to verify that data hasn't been tampered with or altered.

Message authentication includes a key with the hash.

This hash is one-way, we can't decrypt the hash.

5   Public-key crypto

There are four building blocks to cryptography: encryption, hashing, **public-keys**, and random numbers. This section is about public-key crypto.

Key points:

- Whereas encryption has a shared secret known by both parties, public-key crypto has a **public-key** known to everyone and a matching **private-key** known onto to a single party.
- There are three forms of public-key crypto: **asymmetric encryption, key exchange/agreement**, and **digital signatures**.
- There is an extensive **public-key infrastructure** in place to verify who owns which public-key, using **certificates**.

5.1   Public-key encryption

There are three forms of public-key crypto: **encryption**, key exchange/agreement, and digital signatures.

Normal encryption uses the same *shared secret* key known to both parties, the sender and recipient. The same key that encrypts a message will be the key that also decrypts a message.

Through clever mathematics, we can instead use two related keys, such that a message encrypted with one key can only be decrypted with the other key. This allows one key to be made **public**. The other key is kept **private**. This private key is not shared with anybody else.

To understand why this is useful consider how this works for encrypted emails[18]. With normal encryption, you'd need a unique key for everyone you wanted to exchange emails with. You'd have to figure out how to securely share those keys. Nobody could send you an encrypted email without first contacting you and getting a key.

With public-keys, however, you simply post your key to your website, your Facebook bio, to a public directory, and so forth. At this point, complete strangers can start sending you encrypted emails out of the blue. Indeed, they can post those messages on public forums, encrypted with your public-key. Only you with your (matching) private-key can decrypt the messages.

Public-key encryption algorithms are extraordinarily slow. Therefore, they aren't used to encrypt long messages. Instead, we use a normal encryption algorithm like AES, using a randomly generated key. This AES key is then encrypted with the public-key. The recipient in this example uses their private-key to decrypt the AES key, then decrypts the message using AES. We call the use of AES here **bulk-encryption** to distinguish it with the public-key encryption of a small amount of data.

---

[18] There are two standards for encrypted emails, **S/MIME** used by corporations and **PGP** used by crypto-activists.

The most popular algorithm for public-key encryption is known as **RSA**, based on the initials of its creators, Rivest-Shamir-Adelman.[19]

With RSA, the private-key consists of two **large prime numbers**, and the public-key consists essentially of those two numbers multiplied together. The difficulty in cracking RSA is **factoring** that public number back into the original private numbers.

Factoring numbers is different than brute-force cracking of keys. It takes roughly 24 times[20] as many bits for an RSA key to have the **equivalent security** of a 128-bit AES key (that is, 3072 bits). These days, most RSA keys are between 2048 and 4096 bits.

A newer alternative to RSA is based on **Elliptic Curves**, also known as **ECIES** or *Elliptic Curve Integrated Encryption Scheme*. Elliptic curve keys are much smaller, 256-bits for an equivalent security to AES-128. They are also faster in some cases. RSA keys need to be generated from large primes, which are hard to calculate, whereas EC keys can often be generated from any random data..

It's at this point that we are going to skip all the math related to large primes and elliptic curves. The short section on math [page 51] has a little bit more detail, but otherwise, if the reader wants more details, they should look to math-heavy texts on crypto where it's better explained.

5.2   Key exchange/agreement

There are three forms of public-key crypto: encrypted messages, **key exchange/agreement**, and digital signatures.

When two sides interact, they first must agree upon the shared secret they will use for bulk encryption. In the above section, we describe this in terms of encryption, when messages are sent one-way, from sender to recipient. The more general case is when both parties are interacting live with each other, sending packets back and forth both ways, such as in the case with SSL or SSH.

In this general case, instead of one side generating the shared secret and then encrypting it with private-key, both sides generate random numbers and both send them over the public medium. Through this process, they come to agreement on the key without eavesdroppers being able to discover it. This is know as **key exchange** or **key agreement**.

Whereas emails is the best way to understand public-key encryption, SSL is the best way to understand key-exchange. It is (largely) the first thing that happens on SSL. In the first packet[21]. That this can happen quickly, in just two packets going back and forth, is critical to the design of SSL, to minimize the latency.

---

[19] Note that this acronym can lead to confusion. They formed a company in the 1980s to commercialize their technology and patents, and it's still a leading cybersecurity company today. That company puts on a yearly conference known as "RSA Conference". Thus, the letters *RSA* can refer to the algorithm, the company, or the conference, depending on context. In this text, we use it almost exclusively to refer to the algorithm.

[20] As keys get bigger, this number increases. It's 24x given 3072-bit RSA keys compared to 128-bit AES keys.

[21] This describes TLSv1.3. In theory, key-exchange can't start until after the server picks one of the ciphersuites offered by the client. In practice, browsers like Chrome assume the server will pick x25519, and includes the start of

The original algorithm we used for this is known as **Diffie-Hellman** or **DH**. Like RSA encryption described above, it was invented in the 1970s and is based on the math of **large numbers** (around 2048 bits in size). Like RSA, it's being replaced in modern crypto with smaller/faster elliptic-curves, which is called **Elliptic Curve Diffie-Helman** or ECDH.

This text does perhaps a disservice to the reader by describing public-key encryption first and key-exchange second. I do this because you can also do key exchange the same way as with encrypted messages: one side creates the shared secret and encrypts it with the public-key of the other side.

However, it's actually key-exchange that came first. DH was invented before RSA, Elliptic Curve DH is the basis for Elliptic Curve encryption.

An important property that key-exchange has over public-key encryption is **ephemeral keys**. With encryption, if the private-key is every discovered, then everything encrypted with the public-key can now be decrypted. With key-exchange algorithms, inherently, new keys are generated with every key exchange. With SSL, you'll often see the acronym **ECDHE**, which means elliptic-curve Diffie-Hellman with ephemeral keys.

Where this happened in practice was the **Heartbleed** vulnerability, where OpenSSL had a bug that allowed attackers to grab random bits of memory from a server.[22] This successfully grabbed the RSA certificates, and hence allowed decryption of all the SSL sessions. This would not have been practical had ephemeral keys been used instead.

The use of ephemeral keys is also known as **forward security**. Using the old scheme of RSA for key exchange, an attacker can capture traffic first, then later decrypt it once they hack the server and discover the RSA keys that were used. With forward security, this isn't possible, the ephemeral keys are discarded. You can't hack the server and go back in time to decrypt old traffic you captured.

Part of the changes in the latest version of SSL (TLS v1.3) is the use of ephemeral elliptic curve keys (ECDHE) as the preferred method for key exchange.

5.3   Digital signatures

There are three forms of public-key crypto: encryption, key exchange/agreement, and **digital signatures**.

As you'll recall from the sections above, you can create a unique fingerprint using a hash algorithm like SHA2. If you then encrypt that hash with your RSA private-key, everyone else can verify your signature using the matching public-key. This is the basis for digital signatures.

Such signatures are all around us. Most email these days is signed, using a system called **DKIM**, to protect against spam and phishing. We know that emails claiming to

---

this negotiation in the first packet. If the server picks a different ciphersuite, then this will be ignored and have to start all over again.

[22] A tool that efficiently scrapes a server using Heartbleed to grab certificates is https://github.com/robertdavidgraham/heartleech

be from Gmail actually are from Gmail, because Google makes sure all their outgoing emails are signed. This is normally invisible to the user, as part of email headers. If an email lacks such a signature, it'll likely get filed into your spam folder.[23]

Bitcoin transactions are all signed. Your "wallet" is simply your private-key, and your "address" that people send coin to is composed of your public-key.

Where such signatures are especially prominent is in SSL. When you get those website warnings that something is broken, it's usually because of SSL digital signature failures. Web browsers have a list of built-in "authorities" that digitally sign valid owners of websites – when those signatures are invalid, SSL refused to connect. This process is described in more detail in the SSL section below [page 64].

Like with everything else, there are multiple algorithms for digital signatures. You'll be asked to choose which one you want when generating SSL certificates or SSH keys. RSA and DSA use large numbers (around 2048 bits) while Elliptic Curves use smaller numbers (around 256 to 512 bits).

- **RSA**, which simply uses the RSA private-key to encrypt the hash so that it can be verified with a matching public-key.
- **DSA** or **Digital Signature Algorithm**, based on the same math as Diffie-Hellman.
- **ECDSA** or **Elliptic Curve DSA**, an elliptic curve version of DSA.
- **EdDSA**, a variant of DSA using different Elliptic Curves known as Edward Curves, in particular, Ed25519.

5.4   Trust, PKI, and certificates

A digital signature simply punts the issue. It proves the owner of the key owns the document being signed, but it doesn't prove who owns the key. Anybody can post a PGP key for my email address, claiming to be me. Anybody can create their own certificate for a web server claiming to be "www.facebook.com". How do we know the owner is who they claim to be?

There are multiple ways that **trust** might happen. Some of these are:

- **Out-of-band**, such sending confirmations through phone SMS text messages, or copying files manually.
- **Trust on first use**, whereby the key is remembered after the first time it's used and trusted thereafter, such as with SSH.
- **Opportunistic encryption**, which ignores the problem, but assumes anybody intercepting traffic this way will eventually get discovered, such as when nation states get caught intercepting email.
- Some other verification, such as starting an encrypted phone call by speaking aloud words verifying the key, where presumably the other person can recognize your voice.

---

[23] *Yes we can validate the Wikileaks emails*, example using DKIM. https://blog.erratasec.com/2016/10/yes-we-can-validate-wikileaks-emails.html

- A **chain of trust**, where somebody else verifies identity with a digital signature, using their public-key to sign a document (**certificate**) of another person containing their name and public-key.

This section is focused on the last item in this list, which is used by SSL and a lot of other things.

At the root of this is the operating system or web browser. They have a list of **certificate authorities**, companies like "GlobalSign", "Digital Signature Trust", "DigitCert", "Comodo", and so on.

These companies then sign the certificates of **intermediate authorities**.

Finally, these intermediates sign the certificates of websites. In the past, this also used to be an expensive process verifying identities. Sure, you claim to work for Facebook, but how do we know you really are authorized to handle Facebook's certificates? Website certificates could easily cost hundreds of dollars. These days, you can now get free certificates for websites, where identities are verified automatically. A popular intermediate authority that does this is called **LetsEncrypt**. This process is explained in detail in the chapter on SSL [page 68].

If there is a failure in this process, such as a hacker on the network pretending to be a website, your browser will produce an error and refuse to connect.

You can manually check the elements in a certificate chain by clicking on the lock icon, as shown in the image below.
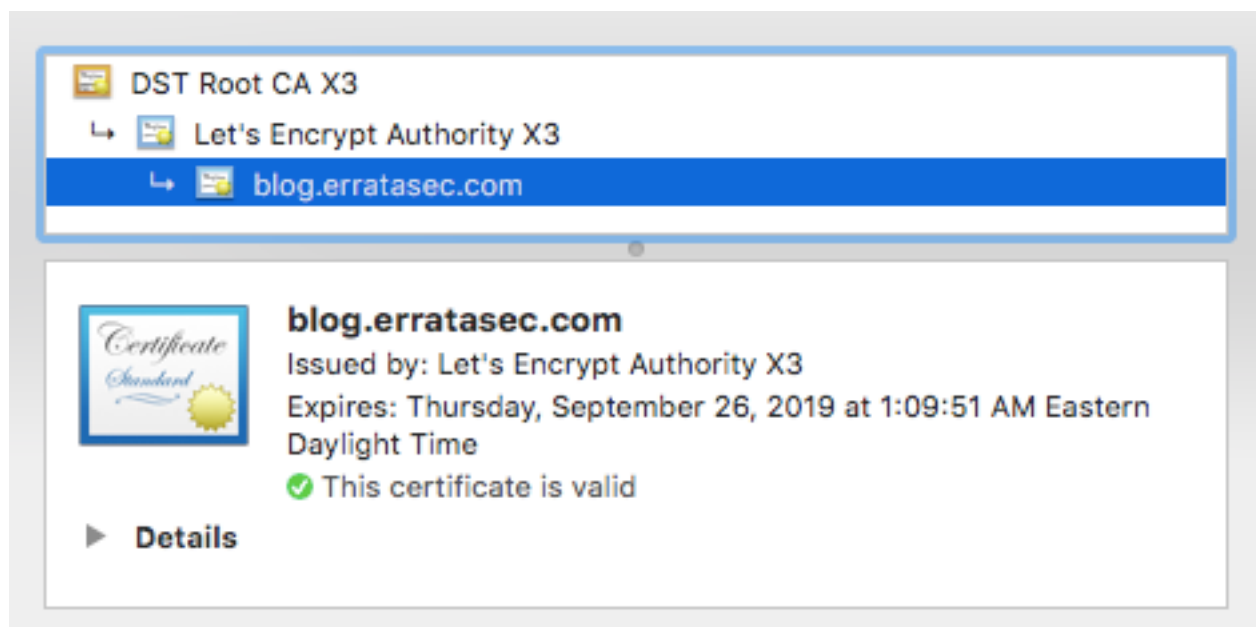
These certificates can be used for other purposes. Large corporations often use something called **S/MIME** for encrypted emails. This involves creating unique certificates for each employee containing the public-key that others can use to encrypt messages to them. These employee certificates are signed by the corporate certificate, which is in turn signed by other authorities, on up the chain to an authority.

Another purpose is signing software, so that you know that it comes from the named vendor, and isn't some fraudulent software containing a virus. For Microsoft, this is known as **Authenticode**. Numerous other techniques exist for other systems.

This entire system is part of the security infrastructure of the Internet and therefore goes by the name **Public Key Infrastructure** or **PKI**.

While PKI is the official system for verifying identity, built into the operating system, there are two alternatives of note.

One is **PGP (Pretty Good Privacy)**, a scheme developed in the 1990s by crypto-activist Phil Zimmerman. It uses a **web of trust** instead of a chain to a root. Individuals use their keys to sign the keys of people they know, like friends and family, whose identity they can personally confirm. In order to trust somebody's identity, you search for a chain of such relationships between somebody you trust and them. For example, your cousin has signed their key, and you trust your cousin, so you trust them. As the "Six Degrees of Separation"[24] theory goes, there should only be a few hops between any two people.

Activists often have **key signing parties** [page 69] where they get together and sign each other's keys. Recently, though, the system has been attacked by hackers going to public key servers and creating zillions of fraudulent signatures for somebody, effectively poisoning their records.

PGP is a popular option for encrypted email among activists, as opposed to the corporate PKI-based S/MIME. Activists prefer this web of trust over central authorities.

Another alternative to PKI is **DNSSEC**, which includes public-keys within DNS records. DNS is the system that translates names (like www.google.com) into numeric Internet addresses (like 172.217.0.132). A *man-in-the-middle* could redirect DNS traffic to themselves and provide fraudulent answers. By including digital signatures, we can validate that DNS hasn't been corrupted or spoofed. In other words, using DNSSEC, we know that 172.217.0.132 is indeed the address for Google, and not some hacker who is messing with network traffic.

Like PKI, DNSSEC has a chain of public-keys. There is a root public-key, which signs the key for ".com", which signs the key for "google.com", which Google then uses to sign "www.google.com".

5.5   Large numbers, elliptic curves, and the quantums

Public-key crypto from the 1977, Diffie-Hellman and RSA, used the mathematical properties of large numbers, which these days are usually around 2048-bits in size. It requires a lot of computation to create the prime numbers, which basically requires generating random bits, testing to see if the number is prime, and then repeating until you find a prime number. There are cheaper ways of creating random primes, but they come at the cost of also being cheaper to brute-force crack the prime numbers. In addition, the algorithms are computationally expensive and slow.

---

[24] https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

Elliptical curves are much smaller, with 256-bit keys the norm these days. They are much easier to create, as pretty much any random data is a valid key. They are computationally cheaper to use.

One problem which this section has largely ignored is that that there are **a lot of different elliptical curves**. There are lots of different curves with different parameters, with various tradeoffs in speed and security when used for different purposes.

There are three main types of curves. The first are those standardized by the governments, such as NIST in the United States. An example of this is the popular **P-256** curve. Another source of curves are those standardize by industry consortiums. An example is the popular **secp256r1** curve (or the variant used in Bitcoin, **secp256k1**). Another source of curves are those popular among academics and cryptographers. An example of this is the **Curve25519**.

Whereas large primes define the past and elliptical curves the present, the future will be defined by algorithms resistant to quantum computers. This is an active area of research, both in the development of quantum computers themselves as well as developing algorithms resistant to them.

5.6   Summary of public-key crypto

There are three classes of algorithms: public-key encryption, key exchange/agreement, and digital signatures.

Public-key, or asymmetric, encryption means that something encrypted with a public-key can only be decrypted by the owner of the matching private-key. It's the basis of encrypted email and messaging.

Key exchange (or agreement) means we can establish an encryption connection across a public channel (where adversaries can watch everything happening), without first knowing any secrets. However, there still needs to be a way of authenticating the other side is who they claim to be.

Digital signatures allow us to sign a document with a private-key that everyone can verify with the matching public-key. Digital signatures is how we (typically) authenticate key exchanges described above.

6   Random number generators

There are four main parts of crypto: encryption, hashing, public-keys, and **random number generation**. Secure random numbers are at least as important as the other parts.

Some key points:

- There are two kinds of random numbers, **pseudo-random** ones generated by mathematic algorithms, and **true random** number (also known as **entropy**) gathered from unpredictable physical events like mouse movements.
- In practice, we usually combine the two, **seeding** the pseudo-random number generator with entropy.
- We often want **predictable** random numbers that we can predict, which the adversary cannot, seeded with a shared secret instead of entropy.
- Programmers need to be careful and avoid the poor random numbers from libraries, which adversaries can predict, and instead get them from the operating system.

6.1   Pseudo-random numbers

Our primary source of random numbers is math, algorithms that generate what's known as **pseudo random** numbers.

If you are a programmer, you are probably with familiar with this. The *rand()* function built into most programming languages gives you random numbers, generating with a mathematical algorithm. However, this well-known generator isn't secure. It's random enough for statistical purposes, but not random enough that hackers/adversaries can't predict the output.

Therefore, for cryptographic purposes we need unguessable numbers that adversaries cannot predict – **cryptographically secure** random numbers. This is actually pretty easy to accomplish, since that's how all cryptographic algorithms are designed, producing output that appears perfectly random to adversaries. Therefore, we can use encryption algorithms (like ChaCha20) or hash algorithms (like SHA2) as the basis for generating random numbers.

Pseudo random number generators contains a small amount of **state** as they run. In poor generators, like the *rand()* function, this state is only 32-bits. In cryptographically secure ones, the state is some large quantity, like 256-bits.

The starting state is known as a **seed**. Given the same seed, the pseudo-random generator always produces the same **sequence** of numbers. It's math. Whenever math is given the same inputs it always produces the same outputs. The only way to get different results from a pseudo-random generator is to give it a different seed to start from.

Another way of thinking about the seed is that it's the key, such like keys used in the other encryption functions, or in keyed hash functions. Just like in those algorithms, the *equivalent security* is the number of bits within the seed.

As well see in the sections below, we seed pseudo-random numbers either with true entropy we get from the outside world, or with shared secrets.

6.2   Entropy and true randomness

Math can't produce true random numbers. **True randomness** or **entropy** comes from unpredictable events in the real world. Examples are mouse movements from the user, arrival time of network packets, or thermal electron drift in silicon chips.

Until recently, the programmer was responsible for writing the software that would gather such information. When generating public-keys, for example, programs would ask the user to move the mouse (or type on the keyboard) to generate random events. These days, however, operating systems do the job for us. Computer hardware, managed by operating system kernels, are hotbeds of random activity, so it's best to let the operating system track gather this entropy and keep it in a pool.

Like everything else in crypto, entropy is measured in bits. We measure how random a thing is by the number of bits we cannot predict.

A good example is the exact timestamp … **NOW!** … as I write this sentence, measuring to the nearest second. How many bits of unpredictability is in this measurement?

You can be fairly certain that I wrote this sometime during the day, so sometime between 8am and 8pm, which is around 40,000 possible seconds, which is roughly 15-bits of randomness. You also know the date I wrote this is one of the four months around summer 2019, which is 120 days, or 7-bits of randomness. Added together, my timestamp has roughly 22-bits of unpredictability.

Let's say I encrypted a file using an AES key based upon the current timestamp. Even though the AES key is 128-bits and thus too big for you to guess, the randomness of the timestamp is only 22-bits, or 4-million combinations. That's trivial. You could easily write a program to brute-force guess all combinations of what that timestamp could be in milliseconds.

That's why you need a more sophisticated source of entropy. Whereas 22-bits can be brute-forced, something like 256-bits cannot be – just like any other cryptographic algorithm (see [page 9] and [page 40]).

To find more entropy, the hardware grabs high resolution timestamps when responding to **interrupts**. A hardware interrupt is when devices like mice, disk drives, and network adapters notify the kernel that data is available. The timing of these events can often be measured in nanoseconds, which often have at least 20 bits of entropy per measurement. That's why moving the mouse around quickly generates enough entropy to generate 2048 bit RSA keys.

The kernel is constantly gathering entropy in the background this way as it runs, storing those bits in a **pool**.

In the past, we'd try to consume entropy directly from this pool, which caused problems when we were emptying the pool faster than it was being filled. These days, we don't read entropy directly. Instead, we use this entropy to *seed* a pseudo-random number generator. As long as the amount of entropy is large (~256-bits) and the

pseudo-random algorithm is cryptographically secure (based on ChaCha20, SHA2, etc.), then the results will be cryptographically secure.

In other words, as long as we start from 256-bits of entropy, and the algorithm has at least 256-bits of state, and the algorithm is cryptographic, then we can produce gigabits worth of pseudo-random numbers confident that all the numbers we produce have 256-bits worth of security behind them. There's no particular need to pull these numbers from a source of true entropy.

6.3   Deterministic random numbers

Up to this point we've talked about random numbers that even we ourselves can't predict, seeded with entropy. In other cases, we want to produce deterministic numbers, seeded with a secret key. In other words, there are two classes of random numbers:

- Random numbers that surprise even us, that we can't predict, from a **PRNG** or **pseudo-random number generator**.
- Random numbers that we can predict, but which the adversary can't, produced by a **PRF** or **pseudo-random function.**

In SSL, we use the PRNG to produce keys and the secrets that we'll exchange with the other side. It doesn't matter what values are produced, only that adversaries can't predict them.

Then in SSL, we need to **stretch** that shared secret for other purposes. We need to generate the key we'll use for the bulk-encryption algorithm like AES. We need to generate the nonce for that encryption. We need to generate a key for the message authentication. We need to get potentially thousands of bits from the mere 256-bits that we've exchanged. Both sides need to derive the same bits from the same key, but the adversary should not be able to figure this out.

Another way of describing the difference is that a PRNG is stateful and a PRF stateless. To get the 10th number in sequence using a PRNG, we have to generate the first 9 numbers. With a PRF, we can just get the 10th number directly.

Outside of SSL, another example of deterministic random numbers is Bitcoin. In Bitcoin, your "wallet" is your private-key, which is a 256-bit elliptic curve for signing transactions. This can be 256 random bits from a PRNG that you store in a file. Or, you can start from a memorable passphrase like "T'was a dark and stormy night" and generate the key using a PRF.

You can use SHA256 as a PRF. You can hash the above phrase to get the hash:

```
849F99222B1C822DFA4A04C781126C4E4858DA6E5EEA82FC24AA5DDC924B13EB
```

Going through the steps[25] to convert this to a private-key, then public-key, then address, creates the following address:

---

[25] There is an online Bitcoin address generator that goes through all these steps at:
http://gobittest.appspot.com/Address

```
1Ee4Kx2wsSmcqN8aWGXJbyERJNAWPRY8qV
```

The point is that this address is predictable. We can discard all this work and 10 years from now you can remember the fact that you started from "T'was a dark and stormy night", and go through these steps again to create your Bitcoin key, and then recover any coins that were sent to you.

This is affectionately known as a *Brain Wallet*, as it's a wallet that exists purely in your head, not written on any paper or stored on any computer (assuming you burn any printout that it was printed on and wipe any disk drive that temporarily held it).

With Bitcoin, you can also generate secondary addresses based on the original private-key. To the outside world, they appear as unrelated addresses that don't belong to you. To your wallet software, though, they all appear as part of your wallet, even though it only retains the one private key.

6.4   Classic random failures

There are three well-known failures of random numbers.

The first was within **Netscape** back in the late 1990s. When exchanging random encryption keys, the random number generator created predictable keys. This allowed somebody to guess the values of SSL keys without cracking either the public-key or encryption algorithms.

The second was within **Debian OpenSSL**. A bug in how the code was optimized removed a critical randomization step, such that SSL certificates were generated with predictable keys. This allowed a ton of websites to be impersonated and attacked via *man-in-the-middle*.

The third example is the **Dual_EC_DRBG** algorithm [page 42], which is widely believed to have been backdoored by the NSA. The NSA pushed this algorithm through the standardization process and paid to have it included in RSA's popular BSAFE crypto library.

6.5   Equivalent security level for random numbers

As in other sections, we evaluate the equivalent security level of algorithms (128-bit AES equivalent to 3072-bit RSA equivalent to 256-bit SHA2). For random numbers, we have similar concerns. How much entropy is enough? How secure does the CSPRNG need to be?

A web server needs to generate a new random 128-bit AES key for each new connection, often tens of thousands of connections per second. I needs millions of bits-per-second of randomness, but if we seed it with a 128-bits of true randomness, we can generate the rest of the bits with a cryptographically secure algorithm. Attackers won't be able to guess the keys by guessing the original entropy, and won't be able to observe a few keys in order to guess which future keys will be *randomly* created.

The same is true if you want to generate a 3072-bit RSA key. You don't need 3072-bits of entropy, just around 128-bits. Furthermore, you can continue creating new certificates without having to gather more entropy, by using a CSPRNG.

When starting up, operating systems will consider that once they have around 256-bits of entropy, they have "enough" for any cryptographic purposes for further use by a CSPRNG.

6.6   Random for programmers

For programmers, getting random numbers is difficult. The standard APIs available to you use an insecure pseudo-random generator combined with an insecure seed (either a fixed value the same every time, or the current timestamp).

To solve this, you need to find a source of true entropy and a cryptographically secure algorithm.

These days, you can get this from the operating system that will do both of these:

- Linux – getrandom()
- macOS/BSD – arc4random()/getentropy()
- Windows – rand_s()

However, these functions only work in the newer versions of the operating systems. Writing code that must work on older systems can be difficult.

For Windows, *RtlGenRandom()* works as far back as WinXP, and is what many crypto libraries rely upon for Windows.

For old Unix systems (most everything not Windows), the device file */dev/random* can be used to get raw entropy. It's not a good source to rely upon if you need a lot of random data, as it will block when the entropy pool is emptied. Newer Unix systems often provide an alternative named */dev/urandom* which is seeded from at least 256-bits of entropy then uses a cryptographically secure algorithm to produce numbers from then on, so does not exhaust the entropy pool.

Both these pseudo-files (*random* and *urandom*) have the flaw that they may not be available in containers and chroot jails, which often restrict a program's visibility of the file system to a single directory.

Programmers may also fail to open these files if they run out of file descriptors. Hackers have been known to deliberately consume all the file descriptors in a program by opening many TCP connections, causing such randomness to fail. Therefore, programmers should use the system calls in preference to these pseudo-files, and only backoff and try to read these files if running on older platform.

If neither the system calls or */dev/(u)random* files exist, then the programmer has a problem. The sample code accompanying this text has a files *util-entropy.c* and *util-random.c* that try to solve this problem by gathering its own entropy. This is only intended for demonstration purposes. If this is a problem, you should probably use a cryptographic library like *OpenSSL* or *libsodium*.

Note that if you mention */dev/random*, you'll probably get people yelling at you to use */dev/urandom* instead. That's because there still is bad blood from when */dev/urandom* first become available, with some people (falsely) believing */dev/random* was more secure because it was true randomness whereas */dev/urandom* is only pseudo-random seeded with entropy. This is wrong thinking.

In any case, you shouldn't use *dev/urandom*, but one of those system. You should only back off and use this pseudo-file if the systems calls (like *getrandom()* or *getentropy()*) aren't available.

In addition to the problem of cryptographically secure randomness programmers have the problem of **uniform** numbers evenly distributed across a range, instead of bunched up on one side or the other. Most random number generators create uniform *binary* numbers, which means any range that is a power of two (1, 2, 4, 8, 16, 32, etc.) will have an even distribution. However, other ranges, like 1-10, will generate biased results.

There are well-known algorithms for generating unbiased numbers from a biased source. The *util-random.c* code accompanying this text shows an example.

6.7   Summary of randomness

There are two kinds of random numbers. The first are ones that you (and your adversary) can't predict, that come as a complete surprise, whose specific values you don't care about. The second are ones that you can predict but your adversary can't, deterministically derived from a seed, so that both you and the party you are communicating with will generate the same sequence.

The truly random numbers aren't, really. We grab only a small amount of truly random data (entropy), enough to be secure, such as 256-bits. Then we use that to see a cryptographically secure pseudo-random algorithm.

Many crypto algorithms are adequate for pseudo-random generators. Two popular ones are ChaCha20 and SHA2.

7   Attacking/breaking crypto

The best way of understanding crypto is understanding the problems it's trying to solve, the attacks it's trying to defend against. For example, SSL would be a simple design if we didn't have to worry about *man-in-the-middle* attacks (see below) – all the complexity with certificates comes from this.

Some key concepts in this section:

- Brute-force, trying all combinations of keys.
- Cryptoanalysis, finding flaws in the algorithms.
- Man-in-the-middle or active eavesdropping of network traffic rather than just passive eavesdropping.
- Key distribution (intercepting keys).
- Implementation flaws.

7.1   Brute-force attacks

We covered brute-force attacks in the section on encryption. A **brute-force** attack is where we try all keys until we find the one that decrypts the message. Computers are now fast enough that algorithms with short keys, such as 56-bit DES, can successfully be brute-forced. However, longer keys, as in 128-bit AES, are far beyond the capability of today's computers.

As explained in the section on encryption [page 9], brute-forcing keys has **exponential difficulty.** That means keys are either easily crackable by even the cheapest computer, or uncrackable even by all the computers in the world working together. There is only a small area in between the two.

This is counterintuitive. Intuitively, we often expect that doubling the size of the key will double the difficulty. That intuition is wrong. Doubling a 40-bit key to 80-bits makes it a trillion times more difficult to crack. Adding one bit, making it a 41-bit key, is what doubles the difficulty.

Exponential growth in difficulty means that short keys are easily brute-forced by even the cheapest computers, but long keys are impossible to brute-force, even by all the computers in the world working together.

Since modern crypto uses long keys, nobody every attacks crypto this way. In other words, since it's the most important attack we worry about, it's the least successful attack in practice.

7.2   Cryptoanalysis of algorithms

After brute-force, the second most important attack is the analysis of algorithms looking for weaknesses, known as **cryptoanalysis**.

Over time, we find more and more weaknesses in algorithms, ways to defeat then in less time than brute-force. Much of the time, these weaknesses are only theoretical, or only shave off a couple bits of the effective strength of the algorithm. In such cases, we continue to use the algorithm. Sometimes weaknesses are found that only apply to certain weak keys, in which case we update how we use the algorithm.

However, in the worst cases, we completely break or **crack** the algorithm, meaning it's no longer secure and can't be used. MD5 and SHA1 are good examples of this – while these algorithms can still be used in certain cases, by and large they've been successfully cracked.

How we replace defective algorithms is a critical part of **protocol** design. When we look later at SSL and SSH, we see that these protocols start by negotiating witch each other which algorithms to use. Over time, we remove weak algorithms from these lists and add strong algorithms.

The most common type of cryptoanalysis is **frequency analysis**. The output of any crypto algorithm should appear as a completely random sequence of bits. If non-randomness can be found, such as certain combinations of bits appearing more frequently than others, then we've found a way to crack the algorithm.

In fact, this has been a popular puzzle in newspapers for over a hundred years. The puzzle consists of a short message encrypted with a **substitution cipher**, replacing each letter with a different one. The most common letters in the English language are ETAON, which you can match up with the most common symbols in the cryptogram. You can likewise exploit common two and three letter combinations, as well as guess short words.

If you'll remember from our example in the encryption section, we used a substitution cipher to encrypt "Attack at midnight" to create the following ciphertext:

```
JBBJMV JB GDZRDIEB
```

This is a little evil of us, as the most common letter in the English language is E, but we don't use it in our message. But the next two common letters are T and A, which do indeed match to the most common letters in the ciphertext. In the middle we have the common two letter word "at", encrypted as JB. Thus, with frequency analysis, we can crack the ciphertext even without the key.

7.3   Known and chosen plaintexts

In our stories of cracking codes, we usually present the problem as receiving an encrypted message with unknown contents, then decoding it. In practice, we usually have more capabilities.

One capability is that we often know parts of the encrypted messages. Our adversary may start all messages with the same greetings, for example. Being able to match the known plaintext and ciphertext together may tell us how to decrypt the rest of the message.

Another capability is that sometimes we can force our adversary to encrypt a message of our choosing, and then from that figure out how to decrypt the unknown messages.

Both of these could be used to crack WEP, the original WiFi encryption protocol. The fact that all packets started with the same, known header help a lot in cracking the WEP key. In addition, knowing their IP address, we could just ping them from the Internet, which when it arrived on the local network would be encrypted with WEP. Since WEP

used a stream-cipher (RC4), this gave us keystream allowing us to decrypt any packet using the same initialization vector.

The lesson here isn't that you'll ever do a chosen/known plaintext attack, but that we often know a lot more about encrypted messages than people realize.

7.4   Side channel

So far, we've talked about attacking the encrypted message itself. Another set of attacks are against the things that process the message, looking at radio emissions, current draw, heat production, or CPU processing time.

This is an important flaw in AES. It uses a substitution table held in the CPU cache. Another thread running hostile software on a CPU can eject elements of the table from the cache in such a way as to discover the AES key.

This flaw is so important that some crypto libraries refuse to do AES in software, but instead will only do AES if hardware acceleration is present. AES hardware, such as in Intel processors, avoid this sand channel issue.

The ChaCha20 stream-cipher is a popular alternative in modern SSH and SSL code because it's efficient in software without side channel issues. It doesn't use substitution tables.

In the last decade, a ton of side channel issues have popped up in modern CPUs, like Intel's processors with "Meltdown" and "Spectre". Stamping out these bugs has taken considerable effort in both operating system and hardware design.

7.5   Oracles

An **oracle** is something whose behavior detectably changes depending on input. This allows an adversary to manipulate the input and test how the oracle responds.

A common one to watch out for is the common "compare" functions found in most programming languages. They typically compare a byte at a time and exit as soon as the input bytes don't match. This allows an attacker to play with the input and measure how long the comparison takes. This has been used since at least the 1970s in order to figure out the right password for a system. Instead of having to guess the entire password, the attacker only had to guess all 26 letter for the first byte until they found the right one, then try the second byte, and so on. A 10 letter password would take at most 260 attempts.

Therefore, in crypto libraries you see "secure compare" functions that take the same amount of time regardless whether or not the two inputs are the same or different.

7.6   Backdoors in algorithms

TODO

There's a way of choosing the default constants such that it'll have a weakness allowing whoever picked those constants to predict the random stream of numbers. The NSA picked the default constants. It's an extremely poor algorithm. There are better options available (such as ChaCha20-based *arc4random()*). Therefore, it's widely believe

the NSA pushed this algorithm specifically so they could also pick backdoored constants.

TODO

7.7   MitM (man in the middle)

Public-key protocols, like that in SSL, create encrypted connections. However, when a hacker is able to redirect network traffic, this only means you've created an encrypted connection to the hacker. This is known as a the **man-in-the-middle** or **MitM** attacks. In order to eavesdrop on the traffic, the adversary redirect traffic to themselves and decrypts it. They then create a new connection to the desired destination, re-encrypting it as it goes outward. Both sides see an encrypted connection, but it's actually two encrypted connections going to the hacker in the middle.

The complicated parts of SSL are from dealing with MitM attacks, having certificates and a PKI infrastructure to authenticate that the server you are talking to is who it claims to be and not some hacker. When the certificate check fails, the web browser knows there's something wrong, and refuses to connect. This is why you get browser warnings when visiting websites with invalid certificates. Of course, most of the warnings you get is because something is broken, not because a hacker is attacking you.

There is some value to encrypted connections even without authentication with certificates. This is known as **opportunistic encryption**. It's currently how email servers often talk to each other, since historically they didn't use encryption at all, and authenticating connections would break too much. When email servers talk to each other (using SMTP) they'll try a command called **STARTTLS** if it's available, and if it is, they'll switch to SSL. But they won't validate certificates. This allows man-in-the-middle attacks which either downgrade the connection preventing SSL from being established, or which provide fraudulent certificates that can't be verified.

It's no worse than what they were doing before, not using any encryption. But more importantly, there is some benefit. Many email programs will warn you whenever the certificate changes, so that if you are checking email on the WiFi hotspot at the airport or local café, you'll have some indication that a hacker may be trying to intercept your email. More importantly, when nation state hackers (like the authoritarian regimes or the NSA) is doing mass spying, we'll notice it. It means they can't sit quietly on the wire passively eavesdropping, but that they must actively participate in the network traffic, leaving traces that we can notice.

Corporations will often MitM their employees. They do so by installing their own root/authority certificates in the laptops/phones of their employees. This allows the MitM devices to create "valid" certificates for every website the employees visit.

A common flaw for mobile apps is that they don't validate certificates. This is either because the apps were poorly written to begin with, or they just go tired of dealing with certificate errors. Either way, it's common for hackers to setup MitM on WiFi hotspots just to go after such apps.

7.8   Implementation errors

As noted above, all crypto these days is so-called "military grade". In theory, it can't be hacked, even by nation states. In practice, crypto can often be bypassed attacking the software that implements it, rather than the crypto itself.

An example is the San Bernardino terrorism case, where the FBI collected the phone from the terrorists and demanded Apple's help in decrypting it. They were eventually able to decrypt the phone without Apple's help, by hiring a firm that had discovered a bug in Apple's software.

The world of crypto is full of examples like this. People regularly misuse crypto, not understanding nuances like this. Thus, despite something like PGP being uncrackable when used correctly, it is used incorrectly often enough as to not impede the NSA's mission.

This has important implications for programmers. If they don't carefully follow the procedures for implementing such things as SSL, taking short cuts or inventing their own solutions, then adversaries will easily defeat them. This is why cryptographers are notoriously conservative, because any creativity results in defeat.

7.9   Eventual loss of keys

TODO

7.10   Key distribution/exchange

TODO

When we discuss encryption, we often talk about how your adversary intercepts the message and tries to decrypt it. In practice, the problem is actually that the adversary intercepts the key, and is able to decrypt all the messages they intercept.

For example, let's say the enemy catches your spy and tortures them to get their encryption key. That means they can decrypt any message they intercepted from that spy.

This is why the most important development in cryptography is probably public-keys, invented in the 1970s. It allows keys to be distributed securely.

Encrypted emails is a the best example. Without public-keys, you'd somehow need to know the key used by everybody that you send email to, a process likely prone to failure. Without public-keys, you don't need to worry about it. You can simply lookup their public-key on a website, and use it. Sure, you have to make sure that it's the right public-key, and not some fake used to man-in-the-middle email, but either PKI or the PGP web of trust (described above) can verify the correct identity of the keys.

Now if they enemy captures and spy and the public-key used to send messages, the enemy still can't decrypt the messages, because they don't have the matching private-key.

7.11 Replay attack

An adversary may not know the contents of a message, but they can still copy it, then send it again. If the message was "attack at midnight", the attacker can replay it a few days later to confuse the enemy into making a pointless attack.

Therefore, protocols need to be designed such that this isn't possible. The most common way is with a **nonce**, such as a high-resolution timestamp with each message that changes how it is encrypted.

7.12 Black-bag and rubber-hoses

TODO

7.13 Online vs. offline attacks

Take, for example, how my accountant recently sent me a statement consisting of a PDF file that was encrypted using the last four digits of my social security number as a PIN number. That means an adversary would need to brute-force 10,000 combinations in order to read the PDF, which can be done with an automated tool in under a second.

So why is a PIN number secure for an ATM machine when withdrawing money, but not secure for encrypting a PDF? The difference is that you can't automate a brute-force crack of the ATM machine. The machine introduces delays between each attempt, and then locks you out after too many bad attempts. A PDF document is unable to enforce such protections.

8   Basic principles

8.1   Kerckhoff's Principle

The most common naïve belief in cryptography is that we must keep everything secret. If I ship a product with a super-secret crypto algorithm, then hackers are less likely to crack it than if I make that algorithm public.

The history of cryptography has proven this not to be true. Instead, what happens is that hackers can easily figure out the secret algorithm anyway, and all you have done is prevent your friends from pointing out obvious errors.

This has been true since the 1800s, when this principle was first formulated as *Kerckhoff's Principle*, though it's been independently stated by others over the years.

When organizations claim to have great crypto, but refuse to tell you how, when they hide their algorithms, then it invariably means they have bad crypto. If they had good crypto, then they would proudly display this, so that independent cryptographers could verify that they indeed have great crypto. The only reason to keep details secret is to hide bad crypto.

Kerckhoff's principle is often expressed as the **Fallacy of Security through Obscurity**. You should probably avoid using that phrase. It's basic marketing. When you keep repeating it, people will remember the phrase but not the underlying reasoning. They'll often forget whether it's a bad thing or a good thing, and many will remember this as stating that obscurity is a good thing. This is especially true since that matches their prejudice. We want to instead drill into people the knowledge that obscurity is a bad thing.

This principle is why big companies like Microsoft, Google, and Apple make their security so public, documenting their algorithms, and in many cases, making their code public so that anybody can find faults and notifying them. Apple's developer's guide describing the security of the iPhone is a wealth of information.

This principle doesn't mean that all your source code must be open, though many open-source advocates will make this claim. It just means that transparency leads to more security than obscurity. The security of a system should reside in direct things like secret keys.

8.2   Military grade encryption

As we discussed with the secret strength of encryption, there is either crypto everyone can break, such as your neighbor's teenager using a computer they bought with their baby sitting money, or crypto nobody can break, not even the NSA spending billions of dollars. This is graphed in how long it takes to break keys of various sizes, using a $35 Raspberry Pi, a desktop computer, or a billion dollar's worth of computers:

Time to brute-force key by device type and key length
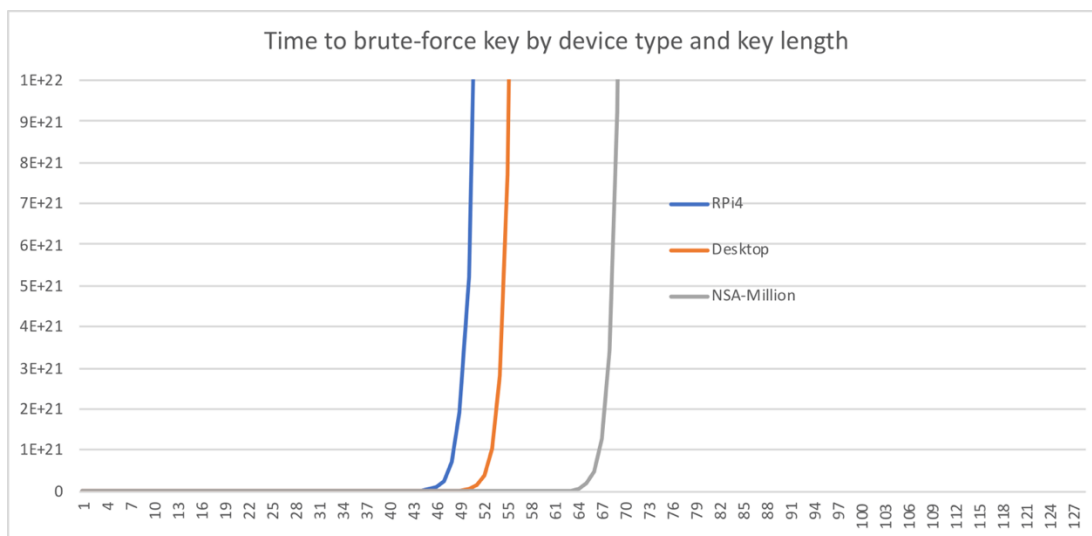
You'll commonly see crypto products advertised as **military grade encryption**. This is red flag that what they are selling is snake oil, since there is no such thing. Everybody these days uses crypto of 128-bits of strength or better, which is equally unbreakable whether it's for consumers or the military.

Indeed, if you took this nonsense seriously, then it would mean that military grade crypto is worse. Consumer products like the iPhone or Google Chrome have the very latest security, whereas military products tend to be decades old. Military grade products are built to withstand the radiation of space or the heat of the Afghan desert. They are designed to be operated by poorly trained soldiers under fire.

An example of "military grade crypto" is the following screenshot from the TV series "Battlefield Afghanistan". This series showed American soldiers fighting in Afghanistan. This particular screenshot comes from them using a computer to download video from a drone flying overhead to discover where the enemy was hiding.

The thing to notice from the screenshot is that the password for the laptop is printed on tape on the laptop. This detail is blurred out through much of the program, but if you step through the videos frame by frame, you'll find some frames where they failed to censor this.

The point of this isn't to laugh at the poor job editing the video, or to laugh at the insecurity of using a simple password composed of the upper-left rows of keys q1w2e3r4, but to appreciate the fact that military security isn't always the best. Security is a tradeoff everywhere, measuring risks vs. benefits, and such measures mean that security for consumers is often higher than for soldiers.

8.3   Don't invent your own algorithm, protocol, or implementation

Crypto is hard, really hard. Even cryptographers can't get it right. This means that we should be extremely conservative, learning how to use existing designs rather than inventing our own designs. That's largely the purpose of this text, to teach programmers how to use the solutions provided by crypto libraries, like *OpenSSL* or *libsodium*, so that developers won't come up with their own designs.

However, we observe naïve programmers constantly inventing their own crypto, including algorithms, protocols, and implementation. It's based on a number of fallacies. One fallacy is the one mentioned above, that if the details are kept obscure, then adversaries won't be able to discover the weaknesses. Another fallacy is that if it's so complex the inventor can't figure out how to defeat it, then others won't figure out how to defeat it either.

There is often the misconception that well-known designs are too complex. This is sometimes true, as cryptographers are horrible coders, and often make solutions that non-cryptographers are unlikely to understand. But in many other cases, such as OpenSSL-style libraries, the only complexity is necessary complexity. They have the minimal number of steps to be secure, and that if you bypass a step, it'll be insecure.

There is also the opposite problem, the belief in byzantine security. They often create overly complex designs claiming that "taking security seriously" requires such complexity, and that removing the unnecessary complexity they invented will make the system less secure.

These designs, even the byzantine complex ones, are often easily defeated by hackers. The truth is that programmers rarely "invent" anything new, but instead simply repeat known, broken designs.

It's a sort of Dunning-Crypto, where naïve programmers are too naïve about crypto to truly appreciate how naïve their designs are. We see this a lot in cryptocurrencies, such as "Iota", where the designers have created shockingly byzantine yet insecure designs.

The solution is to stick with known designs, like SSL. If the well-known solution doesn't fit the problem, don't try to change the solution to fit the problem – change the problem to fit the solution. If two computers connect to each other over the Internet, there had better be a good reason why it's not SSL. Such good reasons exist, as we'll show when discussing SSL, but they are the exceptions not the norm.

With all this said, you are still going to make mistakes, and you shouldn't feel guilty at this. Cryptographers aren't out there writing a lot of code, programmers are. Just because cryptographers have written code for a problem doesn't mean anybody can use it in practice. You are going to have to solve problems where you are not an expert and where no expert is available. You are going to end up inventing something broken where better known solutions exist, but which you didn't know about.

That's where other principles come into play, such as *Kerckhoff's Principle*. Make your design open enough that experts can discover and point out flaws, then have a disclosure policy where you can receive such information and fix things. Sure, they'll call you can idiot for not being a smart as they are, but don't worry about it.

In some cases, that your crypto is bad may not matter. There are increasing nonsensical requirements for "encrypted" data (such as in HIPAA, a healthcare privacy law) where only a basic solution is needed. It has to exist, but it doesn't have to actually work. Bad crypto is perfectly adequate for such bad requirements.

The upshot is that try to follow well-known solutions for well-known problems, such as SSL, but more importantly, accept that whatever solution you come up with is likely flawed and will need fixing in the future.

8.4   Indistinguishable from randomness

The output of crypto algorithms should be indistinguishable from purely random bits, no matter how non-random the input.

For example, let's encrypt with the ChaCha20 algorithm a message consisting of zeroes, where the key is set to zero, and the nonce is set to zero. Everything is zero. The output of the algorithm is this (in hex):

```
76b8e0ada0f13d90405d6ae55386bd28
bdd219b8a08ded1aa836efcc8b770dc7
da41597c5157488d7724e03fb8d84a37
6a43b8f41518a11cc387b669b2ee6586
9f07e7be5551387a98ba977c732d080d
cb0f29a048e3656912c6533e32ee7aed
29b721769ce64e43d57133b074d839d5
31ed1f28510afb45ace10a1f4b794d6f
```

This is indistinguishable from random data, no matter what tests for randomness that you apply to it. If somehow you can find some non-randomness in this data, you will have broken the algorithm, and you could publish the paper and garner lots of fame.

That's why when we discuss random number generators that we find algorithms like ChaCha20 and SHA-256. If those algorithms are done right, then they should be usable for secure random numbers. They are, so they are.

As we saw with hash algorithms like SHA-256, the slight change in the input will result in completely different output, which we can see when hashing "a", "b", "c", and so on:

```
"a" = 0cc175b9c0f1b6a831c399e269772661
"b" = 92eb5ffee6ae2fec3ad71c777531578f
"c" = 4a8a08f09d37b73795649038408b5f33
"d" = 8277e0910d750195b448797616e091ad
"e" = e1671797c52e15f763380b45e841ec32
"f" = 8fa14cdd754f91cc6554c9e71929cce7
```

We often call this the **avalanche effect**, where the slightly change in any input bit quickly spreads to affect all the output bits. We see this in the Bitcoin block chain, where a million blocks later, if any bit is changed in an early block, this result will be reflected in all the subsequent blocks.

8.5   The crypto lifecycle

Crypto is constantly changing. Whatever algorithms, protocols, or implementations that we have today will be replaced sometime in the future. We find weaknesses in algorithms, which is why we had to replace SHA1 with SHA2, after fatal weaknesses were found. We find other improvements, such as how Elliptic Curves have shorter keys and faster processing than the large primes used in RSA. We discover new requirements, such as the ephemeral keys and authenticated encryption in modern SSL that was missing from older SSL. We discover flaws, such as how compressing data

before encrypting it led to flaws that required us to remove compression as a feature in SSL.

One popular solution in the past was to make crypto pluggable, such as how SSL negotiates a list of ciphersuites at the start of a connection. In such cases, the client provides a list of options it supports, and the server selects among that list an option that it supports.

However, we've found this to be an anti-feature, because of **downgrade attacks**, where the attacker forces an otherwise secure client and server to choose an insecure option. The latest version of SSL removes all sorts of old, insecure crypto algorithms. New designs, such as Bitcoin, often have the crypto algorithms built-in. If they are ever discovered to be flawed, the solution will be to upgrade the entire code base, to prevent the flawed solution ever being an option.

In the long run, we have to also contend with Moore's Law. Computers keep getting faster. Eventually the 128-bit keys we use today will be brute-forced, and we'll have to move to 256-bits. This day may come sooner than expected if there's a breakthrough in quantum computer, though to be fair, we've been expecting an imminent breakthrough in this area for 20 years that hasn't happened yet.

9 Some math

This document promised no math, but it lied. Understanding the Boolean math *xor* is pretty darn important. Beyond this, you might have questions about some things, like elliptic curves, so this section will provide a brief overview. Everything but *xor* can be skipped.

- Binary math, such as **xor** and **rotate**.
- **Substitution** as an operation.
- Some notes about **modular** arithmetic and **prime numbers**.
- Some notes about **elliptic curves**.

9.1 xor

Your view of math is that of normal numbers, with operations like addition, subtraction, multiplication, and division. There is a separate set of mathematic operations for binary numbers, those with a value of **1** or **0**, with operations with then names **and**, **or**, **xor,** and **not**. We also call this **Boolean** arithmetic operating on values of **True** or **False**. I suppose if this were some abstract philosophy course we'd talk about this in terms of Boolean values, but for this text, all we care about is binary.

While these operations are defined as two input bits, we can also apply them iteratively on large binary numbers. We apply the operation on the first bit of both numbers, the second bit, the third bit, and so on for all bits. For example, the following shows applying **xor(a,b)** to two 32-bit numbers. The operation is performed for each column independently of the other columns.

```
     a = 00100100011001111000100011001011
     b = 01010101011111110001101010100101
result = 01110001000110001001001001101110
```

You'll find all these binary operations in crypto, but the most important one is **xor**, represented by the symbol ⊕. It has a number of unique properties.

There are two properties that get us excited by *xor*. The first is that if you take a binary number, xor by a random number, the result will also be random. For our purposes, a "random binary number" is a large number (like the 32-bit number shown above) where roughly half the bits are set to 1 and the other half set to 0.

For example, if the input is a number with all 0 bits, then the xor against a random number will exactly equal that random number. Or, if the input number is all '1' bits, xoring against a random number will equal that random number with every bit flipped. Either way, the result of xoring non-random data with a random number creates a random number.

This is the basis of the **stream-cipher** described in the encryption section. It works by generating a stream of random bits, the **keystream**, then *xoring* that stream with plaintext to produce the ciphertext. No matter how non-random the plaintext, the ciphertext will look random. Any test of randomness will show that the ciphertext has roughly the same number of 1 and 0 bits, not matter what the plaintext originally contained.

Another consequence is in **hash** algorithms. Applying *xor* to two numbers never loses information. If I take a terabyte of data, divided it into 32-bit numbers, and simply *xor* all those numbers together, the result will be 32-bit random bits, and any flipping any input bit will cause a bit in the output to flip, thus proving there is no loss of information.

The other useful property of *xor* is that doing it twice comes up with the original number. Again, this is useful with **stream-ciphers**, where encryption and decryption are the same operation: *xoring* with the keystream. The first time you *xor* with the keystream, you convert the plaintext into ciphertext. The second time, you convert the ciphertext back to plaintext. *Xoring* with the keystream twice means the data is unaltered.

9.2   Substitution

Instead of a simple math operation (addition, xor), we can instead grab a few bits at a time and use them as a lookup in a table to get a replacement set of bits. This is called **substitution**. From one point of view, it's just like a mathematical operation. From another point of view, it breaks any mathematical relationships.

Therefore, when you think of what math goes into crypto, there's three kinds: traditional math, binary math, and these substitutions.

However, we've recently come to view these substitution tables to be dangerous for crypto. That's because in software, other things running on a computer can interfere with memory lookups. This has led to successful ways that one piece of software can discover the encryption keys used by another piece of software.

That's why you'll see the ChaCha20 algorithm used often in SSL as an alternative to AES or Triple-DES. Those algorithms use substitution tables whereas ChaCha20 uses pure math. The point here is that when you see alternatives like this, there's always some hidden story behind them. Cryptography is full of these sorts of things. Why not just use AES everywhere to the exclusion of all else? Well, there are reasons, and while they seem pretty theoretical and remote, people have proven actual successful attacks.

By the way, on most computers, AES is done in hardware instead of software, so it's not always a danger. However, sometimes it's not. For example, the $35 Raspberry Pi was too cheap to pay the extra pennies to license AES hardware acceleration so does it in software instead, leading to this danger.

9.3   Modular arithmetic

This section is completely useless. You don't need to know anything about modular math. But it appears often, so people might have questions. This section tries to do a non-math introduction to this math.

When we described the Caesar Cipher, shifting letters in the alphabet, we mentioned the fact that when we shift letters past the letter 'Z' we wrap back on around and start with the letter A. In other words if we shift the letter 'Y' by 3 positions, we get the letter 'B'.

This is known as **modular arithmetic**. Our Caesar Cipher is defined as *addition modulo 26*. When we look into the math of crypto, we find that such modular arithmetic is used everywhere.

Arithmetic using binary numbers is also inherently modular arithmetic. Adding or multiplying 32-bit binary numbers may result in an overflow, requiring 33-bits to 64-bits to hold the result. This is usually unwanted behavior in programming languages. In crypto, this is explicitly defined behavior

Consider the line of code in a program:

```
x[a] += x[b];
```

Normally as a programmer you'd read this as being something that is not expected to overflow. You assume the two sides are numbers safely small enough. But in a crypto algorithm which randomly sets all 32-bits in the number, it'll overflow about a quarter of the time (when the high-order bit of both numbers are 1). This is expected in crypto.

Thus, when you see a simple *addition* in crypto code, like a plus sign + applied to 32-bit integers, it actually means **addition modulo 2^{32}**.

In these two examples (alphabets and 32-bit numbers), modular arithmetic is just an artifact of how things are done. In other cases, it's the **weird mathematical properties** of modular arithmetic from which we obtain public-key crypto.

For example, some public key algorithm rely upon this equation that you learned in high-school math class when simplifying polynomials:

$$x^{a^b} = x^{ab}$$

When doing this with modular arithmetic, **modular exponentiation**, this becomes the Diffie-Hellman key exchange. Doing the math, figuring out why it works to agree upon a shared key, is actually basic high school math. Figuring out why it can't be broken takes a bit more complicated math they didn't teach you in high school.

You'll often see the expression **mod p**. This means the math is being done modulo some prime number. This has a whole new set of interesting properties.

Elliptic curves are calculated *mod p*. In the next section we are going to cheat and show nicely drawn curves, but that's because we leave off the *mod p* part. When we include it, they don't graph as pretty curves for us humans, but just a bunch of dots on the graph. They are still mathematically "curves", though.

In summary, modular arithmetic was formulized around 1800, but it's been an essential part of crypto since the ancient Romans and Greeks. It's not too difficult to understand the basics, they cover it briefly in high school math. High school math is all that's really needed to understand why things like Diffie-Hellman and RSA work, though it takes a lot more than that to understand why they are secure.

9.4   Elliptic-curves vs large-integers

The public-key crypto invented in the 1970s, RSA and Diffie-Hellman, was based on very **large-integers.** In the 1990s an alternative was invented using **elliptic-curves** that are much smaller and faster to calculate. Elliptic-curves have largely supplanted large-

integers in modern public-key crypto, with large-integers largely retained because of backwards compatibility. In addition, the math of large-integers is easier to understand with a high-school education whereas elliptic-curves are more black magic. The recent TLS v1.3 standard has really come down hard on removing the older large-integers, moving to elliptic-curve algorithms instead.

Note that the third generation of public-key crypto is on the horizon that will someday replace elliptic-curves, that are resistant to quantum computers, but that's another decade away.

To appreciate the difference in size, this large integer is an RSA public-key[26]:

```
25394135189383828314790862400272216068154966502777704775410589587001655999539737823298227957861283875304173763803286631890203454507054404619900376467343998522031407053779051794522629857588253654701460711659369392380182636932906620367919705579687620158236075740914554275397793416285605196508626722564555365835358411847539609731772018743598435660418586141263337302189219305372959335222278369471276395505969859047115609467318053667055908411894248568572362889176571853460226733619876912487250743942837962362397148979357305994205346180740503032833836314425269513158601350443798674070411585044714427806516497538735004442395
```

In contract, this smaller (albeit still big) integer is an elliptic-curve public-key[27]:

```
560752075261540843135679992582047026818220217191186449157221797625339832696891069792041837242953814795593991263554041035287057456696433543063415396997
59822
```

The math of large-integer public-key crypto is based on **exponentiation**, this mathematical equivalence that you learned in high-school when simplifying polynomials:
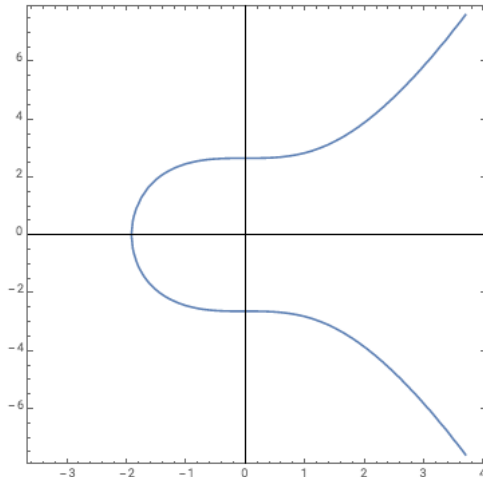
$$x^{a^b} = x^{ab}$$

The alternate math of **elliptic-curves** is based on the polynomial:

$$y^2 = x^3 + ax + b$$

When you graph this equation, you get something looks like the following:

---

[26] Generated with the command *openssl genrsa.*

[27] Generated with the command *openssl ecparam -name secp256k1 -genkey.*

In both cases (large-integers and elliptic-curves), these calculations are done as modular arithmetic, as described above. The above picture of a curve is a lie, because when done with modular arithmetic, it appears as a bunch of random points spread across the plane rather than a nice curve. It's still a "curve" as far as the math is concerned, just not visually so.

Whereas there's (essentially) only a single Diffie-Hellman algorithm, or a single RSA algorithm, there are many curves, with different parameters and features. You can get a list of one supported by OpenSSL by running the command *openssl ecparam -list_curves*. Some examples

- NIST (US Government National Institute of Standards and Technology) with names like P-256.
- SECG, a consortium of crypto companies like RSA and Certicom, with names like secp256k1.
- Informal standards, like Brainpool or Curve25519.

There is a lot of overlap between these, such as NIST standardizing many of the SECG curves.

One of the choices of curves is whether the NSA has backdoored any of them. Most famously in the Dual_EC_DBRG controversy, the NSA is widely believed to have chosen parameters for some crypto algorithms in secret that give them a way to crack the crypto. Many popular curves have arbitrarily chosen parameters with no explanation how they were chosen, which many worry are a exactly such a backdoor.

Bitcoin uses a curve known as *secp256k1* apparently for this reason, having parameters that people believe weren't backdoored by the NSA. It's good bet they haven't -- there are $200 billion worth of Bitcoin betting that nobody at the NSA knows of some secret backdoor in the choice of secp256k1 parameters. If you worked for the NSA and knew of a backdoor in that curve, you could quit your job and steal a few billions of dollars. NSA employees may be patriots, but that's a powerful incentive.

Another curve is known as Curve25519, created independently by cryptographers. It's pretty trusted, and in my monitoring of network traffic, the one most often used when connecting via SSL.

However, that curve shows another lesson. It was used in the Monero cryptocurrency, but in a broken way that would allow a hacker to generate an arbitrary amount of coins. The bug had to be fixed quickly to prevent the cryptocurrency from failing. The lesson here is that elliptic-curves are particularly subtle. Naïve changes can have big consequences. This has paradoxically resulted in elliptic-curves being more reliable than other algorithms. The complexity has made programmers afraid of implementing their own, so they make fewer mistakes.

10   Representing binary data

An important part of working with crypto is how we **encode** binary data. In this context, the word "encode" doesn't mean "encrypt", but simply transform into some form that is more readable to humans. The two most common ways of doing this are encoding in hexadecimal or encoding in BASE64. Bitcoin uses another technique, encoding addresses in BASE58.

10.1   Binary

Technically, everything in a computer is binary, so we should represent it as a series of 1s and 0s, like so:

```
01010101101011010100010111100101…
```

This is cumbersome, so the first thing we do is divide the bits into bytes consisting of 8 bits each.

```
01010101 10101101 01000101 11100101 …
```

Even this is cumbersome, so we have the more compact methods listed below.

10.2   Hex (hexadecimal)

Ideally, we'd like to represent data with the ten decimal digits 0 through 9. However, this doesn't map cleanly to binary data, so we use another format called **hexadecimal**. A combination of 4 bits has 16 possible values. We represent this with the 10 decimal digits plus the first 6 letters of the alphabet, or:

```
0123456789ABCDEF
```

A byte has 8 bits, so we usually represent a byte as two hex digits side-by-side, followed by a space between bytes. Thus, a sequence of bytes represented in hex might look like the following:

```
cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
```

When you see long strings of hex, your eyes are supposed to glaze over. We don't expect you to read this as easily as English text. Much of the time, we don't expect you to recognize the values at all, but instead expect that you'll compare two hex numbers side by side to see if they match, or copy and paste into Google to lookup what they mean.

10.3   BASE64

Hex is designed to be human-readable, at least, by humans that are programmers. There are cases where we need to send binary data in places where only text is allowed, but it doesn't have to be text that can be easily read by humans.

A good example is email. The system was designed to transmit only text, so things that contain non-text data needs to be encoded into text.

The most common way of doing this is known as **BASE64**. It takes 3 bytes at a time and converts it into 4 text characters. The characters are the upper and lower case letters, the digits, and the punctuation / (slash) and + (plus). The name BASE64 comes from the fact that we are using 64 possible text characters.

If you look at the raw source of an email message, you'll often see chunks of BASE64 data.

For SSL, certificates and private keys are often stored in BASE64 form.

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,2202A26B75188AB2

+xizuCh7fLSCOYnhR9L0Jo4C9ZDa/5EGsb+jgQV2dopXq+2O9PCBdkV9EBgsIEkO
FP/lYDH8f3xdfS3KYdMwv8JpCTRYqlGnszQirCCVJmXQue6MyxxxeC7hpgi1eYAx
Aj/tn9AfFmv6gT8o+ULnooVH/Ig5U2U0T4WRKhHkKxJMhU64U0cprnNrMdmcHZ4l
uiy3SkXw+z6+CzL0NyGIBO+f5asc+C3FmvJTbkkpefrypVswXCMpLY8xHX1K5GC+
uv8Hss1+/uNlvcCCRWe6Lkqkct8w4ZIwmSI5B+mkY8EGShNjp4YIIyCBziwbu5Hx
…
```

OpenSSL generally encodes everything with BASE64, such as when you generate private-keys:

```
$ openssl ecparam -name secp256k1 -genkey
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIBq1jeBS8oGa2tDMkETH0yzk0EeAikT/NQ+B0pdtiMo6oAcGBSuBBAAK
oUQDQgAEKjXCO9P3vwAizGRduxqoTJRtDrzdZGjUZN5LtLMwQBhCVQLQ6z2sEJEO
YWq2bBu920MMn5UuxRPgZQOD5IYulQ==
-----END EC PRIVATE KEY-----
```

To repeat, this form is designed for when computer binary data needs to be represented in text, but not in way that is human readable.

10.4 BASE58

Bitcoin addresses use something similar to BASE64, except that it uses only 58 characters. The reason for this is that some characters can be confusing, such as the letter 'O' and the number '0'. This matters when you expect users to type in addresses.

10.5 ASN.1, BER, and DER

Another *encoding* you'll encounter is known as ASN.1. It's actually a way of *structuring* data instead of encoding it. Things like keys and certificates contain many fields, which may in turn contain sub-fields. Fields have varying lengths. Therefore, there needs to be a way of structuring all these fields.

Adding "-text" to many *openssl* commands will *decode* the structure and show the contents of individual fields. For example, here is an example of using the *openssl genrsa* command to generate an RSA key, and then immediately passing it to the *openssl rsa*

command to read the key and decode the fields. I've truncated the fields with … since, of course, they are annoyingly large integers:

```
$ openssl genrsa | openssl rsa -text
RSA Private-Key: (512 bit, 2 primes)
modulus:
    00:e0:fd:74:2c:8c:04:fe:5a:53:ab:ec:2f...
publicExponent: 65537 (0x10001)
privateExponent:
    00:bb:06:35:22:2a:aa:a3:fc:d5:fc:dd:f2...
prime1:
    00:f3:6b:2f:f7:ff:6c:c3:4f:bb:b1:2d:f3...
prime2:
    00:ec:9e:6d:29:bb:17:ac:81:f7:ef:af:04...
exponent1:
    00:d8:6b:88:6e:9a:2e:7d:48:3a:bc:40:fa...
exponent2:
    05:23:3f:7d:8b:79:6d:1e:79:52:b3:fb:24...
coefficient:
    00:bd:02:46:8a:a9:ff:65:a7:6b:43:26:f7...
```

11   Cipherpunks and cryptoanarchy

TODO

11.1   Backdoored standards

TODO

11.2   End-to-end encryption and backdoors

TODO

11.3   Full disk encryption and backdoors

TODO

11.4   PGP and Cryptowars

TODO

11.5   Tor

TODO

11.6   Bitcoin

TODO

11.7   Anonymous remailer

TODO

12   SSH – secure shell

This text assumes the reader is already familiar with SSH, that they use the *ssh* tool on the command-line to login to remote computers.

12.1   How SSH is used

The most popular use of SSH is to establish a **command-line** session with the remote computer. In other words, the user logs in, types some commands, and gets some results back from their commands. The term stands for **secure shell**, where "shell" means the "command-line", and "secure" refers to the fact that it's encrypted. Back in the old days, with tools like *Telnet* or *rsh*, they weren't encrypted and that was bad.

The SSH tools can also be use for **file-transfer**. In this case, the client-side utility is called **scp** for "secure copy".

A lesser known feature is that it can serve in much the same capacity as SSL to establish a generic encrypted network connection on top of TCP. Like SSL, one can create a VPN on top of this.

12.2   Connection protocol

How SSH establishes a connection is much the same as with SSL.

The first step is for both sides to negotiate the protocol. This is a simple packet containing a line of text, like the following:

```
SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.8
```

Pretty much everything is "SSH-2.0" these days. The text after this point is pretty much ignored and is only used by humans in order to debug connections. With OpenSSH servers, the one you most likely have on your servers, the configuration file will allow you to add some text here, so that when you do a network scan, you can easily tell which server is which. (I use this feature a lot).

After both sides (the client who initiated the connection and the server) exchange these packets agreeing on SSH-2.0, each side sends a packet to the other notifying them of which algorithms they support. These will list.

- **Key exchange** algorithms, often traditional Diffie-Hellman or newer Elliptic Curve (ECDH) with curves like Curve25519 or P-256. If you'll recall from the section on public-key crypto, key exchange is where we establish an encrypted channel across a public medium, without both sides knowing any shared secret to begin with.
- **Pseudo-random function**, usually SHA2. This is the function that will take the exchanged secret and expand it out to the bulk encryption key, the nonce, and the message authentication key. If you'll recall from the section on encryption, we need both a nonce and authentication as part of encryption. If you'll recall from the section on random numbers, we need a pseudo-random function to derive such stuff from the whatever was negotiated in the key exchange.

- **Digital signature algorithm**, which is likely either RSA for old stuff or Elliptic Curve (ECDSA or EdDSA) for newer stuff. We often have choices between the P-256 curve or the Ed25519 curve. As you'll recall from the section on public-key crypto, it's through the digital signature that we'll prove that this is the actual server and not some fake server in a *man-in-the-middle* attack.
- **Encryption algorithm**, like AES 128 or ChaCha20. This may also include the authenticated encryption algorithm, like GCM for AES or Poly1305 for ChaCha20.
- **Compression algorithm**, which historically was usually *zlib* (the same one you see everywhere), but these connections often avoid using compression because it acts as an oracle.

After they've figured out a common set of algorithms to use, they do the **key exchange** and establish an **encrypted** connection. The server proves its identity by sending over its **public-key**, then uses that public-key to **digitally sign** what the client sent.

12.3   Server identity

At this point, we've got an encrypted connection to the server, but we've glossed over how the server proves it's identity. Sure, it uses its public-key to sign something, but how do we know this is the right public key?

In the case of SSL, this is done with certificates signed by certificate authorities, and the list of trustworthy certificate authorities baked into the operating system (or sometimes browser, or both). In the case of PGP, it's a web of trust where hippies go to key signing parties putting their keys into a bowl to be signed.

In the case of SSH, this is done by storing the public-key on the client computer, in a file called **.ssh/known_hosts** in the user's home directory (or **%USERPROFILE%\.ssh\known_hosts** on Windows).

The public-key gets into that file in two ways. One way is that the user manually copies it there, such as on a USB drive from the server. Another manual way is to print it out. For example, you can in theory copy the following key into the file in order to identify my server.

```
scanme.robertgraham.com ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLUE6nyqc0xSjcXkkxKBa9b
N0UcBZAE73Ole0CEmTi5Od5PMNIbH+IU2rd7bx1EBQAIpns8aO/sxBKJJpWFUEuM=
```

The simpler way, and the way it happens 99% of the time, is that it warns you about an unknown key the first time you connect, then adds it to the file. This is known as **TOFU** or **time-of-first-use**. It's risky, but from the attacker's point of view, it's also risky for them. If this isn't the first time the user connects (and it probably isn't), then the user is going to detect that some funny business is going on.

12.4   User identity

The next step is for the server to verify the user's identity.

Compared to SSL, this isn't something that protocol really does. SSL does provide a mechanism for client-side certificates, but that's only used in rare cases. For the most part, SSL assumes that if any user authentication needs to be done, it's the job of some higher level protocol, such as a web page.

For SSH, there are two main ways this can be done. The first is simply that once the encrypted connection has been established, the user types their password. All the typical user authentication mechanisms can work here, such as smart cards and LDAP integration.

The second way is that the user creates a private/public key pair and copies the public-key up to the server. This is done with the tool that comes with *ssh* known as **ssh-keygen**. Then, the tool **ssh-copy-id** can be used to copy the *public-key* that was just generated up to the server. Then the tool **ssh-add** can be used on the local machine to copy the *private-key* to where the *ssh* client can find it.

Thus, the next time the user connection, the client will send over the username, and sign something from the server. The server will use the public-key associated with that username to verify the digital signature.

13   SSL

SSL or **Secure Sockets Layer** is the most common way that devices on the Internet create encrypted connections to each other. There are others (such as IPsec and SSH), but SSL is by far the most common. As the name implies, it can be viewed as a *layer* sitting between the application and the network. The word *sockets* refers to the way the apps have traditionally connected to the network without encryption, *secure sockets* implies it's much the same method, just now secure.

There are two parts to SSL, the part that creates the encrypted connection, and the certificates that authenticate who is on each end. This section focuses just on the protocol, while the next section explains the certificates.

Before SSL starts, a TCP connection is established from the *client* (the side the starts the connection) and the server (the side that receives the connection). This is the layer below SSL.

What SSL does is then go through the **Handshake** process, as both sides sends packets back-and-forth to established the encrypted layer.

After that point, with an established encrypted connection, the application on SSL does whatever it wants to do, with SSL doing little more than simply encrypting/decrypting the data.

13.1   Client Hello

The SSL handshake starts with the client (the side that initiated the connection) sending a **Hello** packet to the server (the side receiving the connection). This is around 600 bytes in size, depending upon the client. Some are a lot less, some are a lot more. It contains lots of optional features that the server may or may not support.

The following is a picture from the a Client Hello from Google's "Chrome" web browser.

```
       Length: 592
    ▼ Handshake Protocol: Client Hello
         Handshake Type: Client Hello (1)
         Length: 588
         Version: TLS 1.2 (0x0303)
         Random: 2e452c4d63cff3f222757223114b65ea5a7a30b69a1cd201...
         Session ID Length: 32
         Session ID: c893aa87f2234ecebc2465eed6a7a32124d7b0f2bddba992...
         Cipher Suites Length: 34
      ▶ Cipher Suites (17 suites)
         Compression Methods Length: 1
      ▶ Compression Methods (1 method)
         Extensions Length: 481
      ▶ Extension: Reserved (GREASE) (len=0)
      ▶ Extension: server_name (len=18)
      ▶ Extension: extended_master_secret (len=0)
      ▶ Extension: renegotiation_info (len=1)
      ▶ Extension: supported_groups (len=10)
      ▶ Extension: ec_point_formats (len=2)
      ▶ Extension: SessionTicket TLS (len=0)
      ▶ Extension: application_layer_protocol_negotiation (len=14)
      ▶ Extension: status_request (len=5)
      ▶ Extension: signature_algorithms (len=20)
      ▶ Extension: signed_certificate_timestamp (len=0)
      ▶ Extension: key_share (len=43)
      ▶ Extension: psk_key_exchange_modes (len=2)
      ▶ Extension: supported_versions (len=11)
      ▶ Extension: Unknown type 27 (len=3)
      ▶ Extension: Reserved (GREASE) (len=1)
      ▶ Extension: pre_shared_key (len=283)
```

Each of these optional *extensions* has a different purpose. Let's examine one of them, the SNI or *server_name* field, which we expand below:

```
   ▶ Extension: Reserved (GREASE) (len=0)
   ▼ Extension: server_name (len=18)
         Type: server_name (0)
         Length: 18
      ▼ Server Name Indication extension
            Server Name list length: 16
            Server Name Type: host_name (0)
            Server Name length: 13
            Server Name: pbs.twimg.com
   ▶ Extension: extended_master_secret (len=0)
```

This indicates that the browser is trying to reach "pbs.twimg.com". The SNI field allows something called *virtual hosting*, hosting many domains at a single IP address. Unless the client tells the server which name it's trying to reach, the server can't supply the correct certificate to authenticate the server. "twimg.com" is twitters graphics/image servers and are usually hosted on a *content delivery network* or *CDN* that's usually located in the nearest large city to the user. Virtual hosting is an essential part of CDNs, using a small number of servers to host a large amount of domain names.

SNI is an example of an optional extension. If it's not included in the Hello, then things won't necessarily break. Much of the time, the server will still deliver the certificate the browser is expecting. Only in specific cases of *virtual hosting* will things break. Conversely, if it is included, but the server doesn't support SNI, then it will be ignored. The server will generate a certificate and respond as if the field wasn't included at all.

We aren't going to examine all the optional extensions here. We'll cover a few more as they come up, but for the most part we are going to ignore most of them. This description was only intended to show the general concept that the Hello supports lots of optional functionality which may or may not be supported on the server.

13.2 Cipher suites

One of the non-optional fields is the list of **cipher suites**. A cipher suite is the combination of encryption, public-key, hashing, and randomness algorithms that a connection will use.

The reason there is a list is that over time, we coming up with better suites, and add them to the list of options. Also over time, we keep discovering weaknesses, and remove them from the list. The client and server will have different code bases, and will thus support different lists of cipher suites. Hopefully there is an overlap, or the SSL will fail to connect.

The client gives a list of ciphersuites in its, and the server picks the best one in its response.

The following picture expands the field from the example above, showing 17 cipher suites proposed by the client (which in this case, is the Google Chrome web browser):

```
▼ Cipher Suites (17 suites)
    Cipher Suite: Reserved (GREASE) (0x3a3a)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
    Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
  Compression Methods Length: 1
```

I've highlight one of them for further explanation, TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256.

The first part, **TLS**, refers to **protocol version**. There have been various versions of SSL over time: SSL v1, SSL v2, SSL v3, TLS v1.0, TLS v1.1, TLS v1.2, TLS 1.3. Weaknesses have been found in anything before TLS v1.0, so they should no longer

appear. We don't see it explicit in the name above, but this is specifically a TLS v1.2 ciphersuite.

The next part, **ECDHE**, means using elliptical-curve Diffie-Hellman (with ephemeral keys) as the **public-key exchange/agreement**. Both sides generate some random bit of *private* data, transform it, and send the transformed bit to the other side. They take the transformed chunk from the other side, and transform it against using their *private* data. In this fashion, both sides arrive at the same result through different paths. An observer eavesdropping on the exchange cannot arrive at that result, though, because it would need to know the private data created by either side. The result of this option is the **shared secret** that both sides now share.

The next part of our ciphersuite is **RSA**, which refers to the **certificate's digital signature algorithm**. This signature will digitally sign the key exchange above, proving the identity of the server, and that we haven't been redirected by a hacker to a MitM server.

The next part of our ciphersuite is **AES-128**, using the standard AES **encryption algorithm** with 128-bits. Going down the list you'll note that other choices we might have is AES-256 with stronger keys, the ChaCha20 algorithm using 256-bit keys, or the Triple-DES algorithm using 112 bit keys. In my tests, Triple-DES is roughly 100 times slower on modern x86 computers, which is why it's at the bottom of the list – it's the least desirable algorithm that should only be chosen if nothing else matches.

The next part of the ciphersuite, **GCM**, tells us both the **mode** and **message authentication** algorithm. Remember in our discussion of block-ciphers and the *ECB Penguin* that we need to know which mode with AES. As you can see from the list above, the two options are either a **counter-mode** like GCM or the **Cipher Block Chaining** mode or **CBC**. GCM is much faster, which is why it's listed ahead of CBC. In addition, encryption needs some form of message authentication to verify that an attacker hasn't changed a few bits. GCM or *Galois Counter Mode* includes this sort of check along with the counter mode operation.

The final part of our ciphersuite is **SHA256**. This is a hash algorithm, but it's not specified here for hashing. You might think it's the hash used as part of the digital signature, but it's not, that'll be specified later when we look at the certificate. You might think it's the hash used for message authentication, but it's not, that's part of AES-GCM. Instead, this hash is defined for use as the random number function.

As described above, the key exchange results in a shared secret that both sides know. Based on that shared secret, they need to derive other data, such as the key used for AES. This is done with the random number function. Based upon some small initial shared secret we can use SHA256 to generate any amount of "random" data that we need. Both sides to use the same agreed upon function and same agreed upon shared secret to derive the same data.

Note that the first entries in our list have shorter ciphersuites like TLS_AES_128_GCM_SHA256. That's because in TLS v1.3 they've been removed from this list and are negotiated elsewhere.

TLS v1.3 requires the use of ephemeral Diffie-Hellman for key exchange, which was only optional on prior version. Instead of waiting to first negotiate the ciphersuite, then start the key exchange later, it just starts the key exchange immediately, using the

key_share extension. This saves a back-and-forth step in the SSL handshake, reducing the latency / time it takes to connect.

```
▼ Extension: key_share (len=43)
      Type: key_share (51)
      Length: 43
   ▼ Key Share extension
        Client Key Share Length: 41
      ▶ Key Share Entry: Group: Reserved (GREASE), Key Exchange length: 1
      ▼ Key Share Entry: Group: x25519, Key Exchange length: 32
           Group: x25519 (29)
           Key Exchange Length: 32
           Key Exchange: b1ba5b2f81d8d9558ce66f09495fdff579ec8d28c2017fc7...
```

Likewise with TLS v1.3, how the certificate works is negotiated separately in a different extension field:

```
▼ Extension: signature_algorithms (len=20)
      Type: signature_algorithms (13)
      Length: 20
      Signature Hash Algorithms Length: 18
   ▼ Signature Hash Algorithms (9 algorithms)
      ▶ Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
      ▶ Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
      ▶ Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
      ▶ Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
      ▶ Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
      ▶ Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
      ▶ Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
      ▶ Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
      ▶ Signature Algorithm: rsa_pkcs1_sha1 (0x0201)
```

This list comes in three parts. The first part is which algorithm is used, RSA or elliptic cures (ECDSA). The second part is parameters about the specific algorithm, like in the case of elliptic curves, whether the secp256r1 or secp384r1 curves are used. The third part is which hash algorithm is used to hash the certificate.

13.3   Free certificates with LetsEncrypt

TODO

13.4   TODO

14   PGP

14.1   Generating a key

TODO

14.2   Getting signed with the web of trust

TODO

14.3   Sending a message

TODO

14.4   Receiving a message

TODO

14.5   Email integration

TODO

15   Password hashing

TODO

Instead of storing raw passwords, computers store the hashes. This is why when you forget your password, IT can't tell you what it is. They don't know. Instead, they can only reset it to a new value.

Because of European privacy regulations, storing plaintext passwords gets companies into trouble. Often, even if they store passwords as hashes correctly, other parts of the system may log the plaintext passwords.

TODO

16   The openssl command-line tool

TODO

OpenSSL is a package that includes both programming libraries but also a command-line tool called *openssl*. We can use this tool for a wide variety of tasks, from encrypting messages to creating certificates. The program has a zillion of options, and they tend to change over time, so we aren't going to cover the program comprehensively here. Instead, we are going to look at some common uses.

16.1   Help

You can sometimes get help by using it as the first parameter, such as:

```
$ openssl help
```

This will get produce several screenfuls of information, so the output won't be displayed here.

You can narrow the help for a given command. In order to see which options you can use for *openssl dgst* command, type:

```
$ openssl help dgst
```

16.2   Hashing

To hash something, use "*openssl dgst*" (where "digest" means "hash"). To get help on how to use this, use "*openssl help dgst*".

In the section on Hashing we described how to hash a password, demonstrated in the following way:

```
$ echo -n "Password1234" | openssl dgst -sha256
SHA256(stdin)=
a0f3285b07c26c0dcd2191447f391170d06035e8d57e31a048ba87074f3a9a15
```

To get the hash for an entire file, do something like the following to get the hash of the file "README.md":

```
$ openssl dgst -sha256 README.md
SHA256(README.md)=
135369e322e7695c2aa9ade62effdbcbd541762871778d041076d6bbe332350d
```

When downloading files from the Internet you'll see that they sometimes also tell you their hash values. You can verify them with the *openssl* tool.

16.3   Encryption

TODO

To encrypt something, use "*openssl enc*". To get help, use "*openssl help enc*" or "*openssl enc -help*". To get a list of encryption algorithms you can use, type "*openssl enc -ciphers*".

In producing the *ECB Penguin* example (see the section on encryption modes), I used the command:

```
openssl enc -aes-128-ecb -nosalt -k Password1234 -in Tux.body -out
Tux.body.ecb
```

The first option, *-aes-128-ecb*, selections the *cipher* used, in this case the AES algorithm with a 128-bit key in ECB mode.

The next two options control the password derivation, how the password ("Password1234") is converted into an AES key. These are explained in the section below on password hashing.

TODO

16.4   Creating a certificate

TODO

Creating a certificate is complicated by the fact that it happens in three steps. These are:

- Generate the private/public-key pair.
- Generate a certificate request
- Get that certificate request signed by a certificate authority, resulting in the actual certificate

You can truncate the last step and *self-sign* the certificate. This will be shown below.

TODO

16.5   Creating secret messages

TODO

You can create secret messages to pass to your friends using the technique shown above, when both of you know the same password. However, a better way is using public-keys, where you send a secret message to your friend using their public-key, which only they can decrypt using their private key.

This is fundamentally how PGP works, but in our examples, we are going to use the openssl command-line tool.

TODO

16.6   TODO