

# INTERNET LAW: CASES & PROBLEMS

James Grimmelmann  
*Professor of Law*  
*University of Maryland*

Ver. 4.0 © 2014 James Grimmelmann



[www.semaphorepress.com](http://www.semaphorepress.com)

Front cover photograph: “[Assembly y y y](#)” by [Rosa Menkman](#), available under a [Creative Commons Attribution 2.0 license](#).

# CHAPTER 1: COMPUTERS

---

The first of the four major themes of this book is how law changes when computers – rather than people – make and enforce decisions. It is arguably *the* central question in all of Internet law. Although he was not the first to focus on the question, Professor Lawrence Lessig gave the most influential answer to it: “code is law.” By this, he meant that computer software (or “code”) could do some of the same work that law ordinarily does in controlling people’s conduct. This chapter explores the idea in three ways: a technical primer on how computers and the Internet work, excerpts from Lessig’s book and a more recent one that builds on it, and a case study on what happens when software goes wrong.

## I. Computer Technologies

This section provides a technical primer on computers and the Internet, with an emphasis on the fundamentals that recur in case after case. As you read the fact section of an opinion, it may help to try to fit the court’s discussions of the particular technologies at issue in a given case into the framework provided here.

### Computers

#### Bits and Data

It’s a cliché to say that computers reduce everything in the world to ones and zeros—but it’s also true. In a very real sense, a modern computer is just a complicated electrical circuit. So compare it to a much simpler circuit: a flashlight. We could say that the flashlight “remembers” one piece of information: whether the switch is on or off. When the switch is on, current flows through the bulb and it lights up the night. When the switch is off, no current is flowing, and the flashlight stays dark.

This is a single *bit*: a piece of information that can be either 1 or 0. A flashlight is a one-bit computer. With two flashlights, we could store two bits, with ten flashlights, we could store ten bits, and so on. A computer uses smaller wires and has many many more of them, but the basic principle is the same. The presence or absence of electric current can be used to represent the values 1 and 0, respectively.

Once you have bits, you can describe anything and everything. Numbers are a good starting point. Take, for example, the number 42. To represent it using bits, we write it down in base 2, also known as *binary*: 101010. (That’s a 1 in the 32s place, plus a 1 in the 8s place, plus a 1 in the 2s place, or  $32 + 8 + 2 = 42$ .) So we need six bits to store the number 42: that’s six flashlights, or six tiny wires inside a computer. For historical reasons, computers almost always group bits into sets of eight, called *bytes*. In this system, 42 would be stored as 00101010. (The first two bits are zeros in the 128s and 64s place; the last six are exactly the same as before.)

Using one byte, therefore, we can represent any positive integer from 0 (that’s 00000000) up to 255 (that’s 11111111). From here on, other kinds of numbers are easy. Larger integers just take more bytes: 46,105, for example, is 10110100 00011001. Negative numbers need one more bit: we could say that 1 means “the rest of this number is positive” and 0 means “the rest of this number is negative.” For smaller numbers, we could

use bits to the right of the decimal point as well as to left of it.\* For really big and really small numbers, we could use bits to store an exponent, just like in scientific notation.

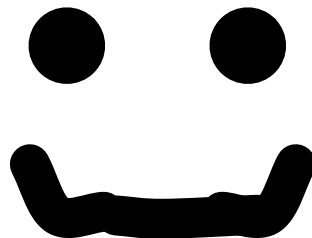
One particularly nice consequence of using binary to represent numbers is that it takes surprisingly few bits to write down even very large numbers. One bit can stand for either of two different numbers: 0 or 1. Two bits can stand for any of four different numbers: 00 is 0, 01 is 1, 10 is 2, and 11 is 3. Three bits can stand for eight different numbers, four bits can stand for sixteen, and so on. Thirty-two bits (i.e. four bytes), are enough to represent any integer from 0 to 4,294,967,295. Adding more bits increases their descriptive power exponentially.

Numbers are useful for calculating, but for communicating, we really need letters and other familiar symbols like @ and \$. The most familiar way of storing, or *encoding*, letters as numbers is a system colloquially known as ASCII.<sup>†</sup> Capital A is 65, capital B is 66, and so on through capital Z, which is 90. Lower-case a is 97, lower-case b is 98, and so on again. ASCII uses other values for other commonly used characters: such as 32 for a space and 64 for the @ sign.

Using ASCII, it's possible to convert arbitrary text into bits and vice versa. Each of the numbers it uses is small enough to fit in a single byte. So, for example, capital A is 01000001 in bits. "Hello!" would be the numbers 72 101 108 108 111 33, or, in bits, 01001000 01100101 01101100 01101100 01101111 0100001. When you open an email containing those bits, your computer converts them back into letters and displays "Hello!" on the screen.

ASCII is the most familiar way of storing text in a computer, but hardly the only one. The Unicode standard—an ambitious system for representing every writing system in use by humans, from Tibetan to emoji—defines an encoding named UTF-8, which represents each character using as few as one or as many as four bytes. (Common characters like Q take one byte; uncommon ones like 🐼 take more.) If you've ever received an email liberally decorated with â€œ and similar gibberish, you've seen what happens when different encodings collide. Someone sent you a message in UTF-8, but your computer interpreted it as ASCII. The same bits—11100010 10000000 10011100—stand for “ when interpreted as UTF-8 but for â€œ when interpreted as the most common version of ASCII.

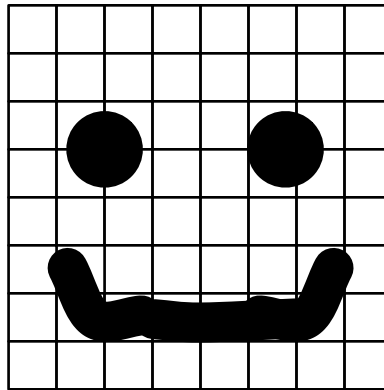
What about images? Suppose we wanted to store this admittedly crude picture of a face:



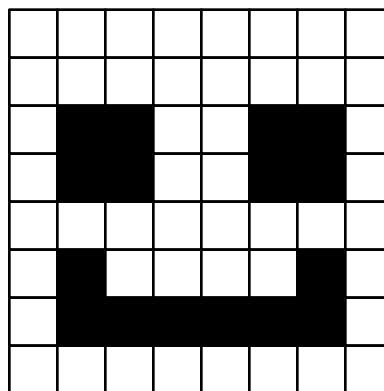
\* Useless Fact #1: technically, the term should be “radix point.” It’s not a “decimal” point because we’re not in base 10 anymore.

<sup>†</sup> Useless Fact #2: ASCII is short for “American Standard Code for Information Interchange.” There are actually numerous variations on ASCII; this section uses the most familiar one, which goes by the official name “Windows-1252.”

Once again, the strategy is to look for a way of breaking something down into bits. We start by overlaying a grid on the face:



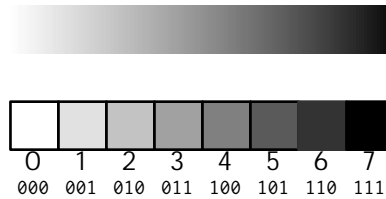
Now, for each box in the grid, we ask whether the face contains more black or more white. The boxes that are more than half black we make *all* black; the boxes that are more than half white, we make *all* white:



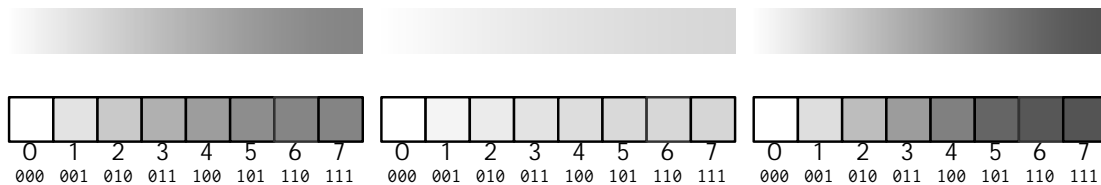
Every box, or *pixel*, is now either all-black or all-white. If we call each black pixel 0 and each white pixel 1, *each pixel corresponds to a single bit*. If we read off the first row of pixels, we get 11111111, since every pixel is white. The second row is the same, but the third row is 10011001 because the tops of the eyes show up as 1s. If we repeat for each row of pixels, the entire face is 11111111 11111111 10011001 10011001 11111111 10111101 10000001 11111111. There you go: an image has been turned into bits, suitable for storing in a computer. To be sure, the drawing has become even cruder. But by using a smaller grid with smaller pixels—for example, by using a camera with more megapixels—we can smooth out the edges until the difference has become unnoticeable.

Interesting images are rarely black-or-white. But another approximation lets computers represent shades of gray. Instead of using a single bit for each pixel, use several. Here, for example, is a three-bit way of encoding the brightness of each pixel. We divide the spectrum from white to black into eight evenly-spaced shades, so that the difference in brightness from each one to the next is the same. Then, we can approximate any shade between black and white by finding it on the spectrum and picking the one out of the eight colors that is closest to it in brightness. All that remains is to convert the eight rep-

representative colors to bits: 000 is all white, 001 is a very light gray, 010 is a slightly darker gray, and so on through 111, which is all black.

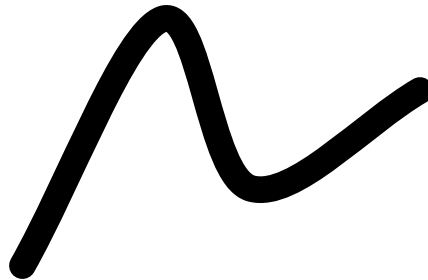


A similar technique makes it possible to represent colors. All colors are made up out of a combination of red, green, and blue light. So instead of encoding a color using a single intensity value, we can encode it using three, one for each primary color:

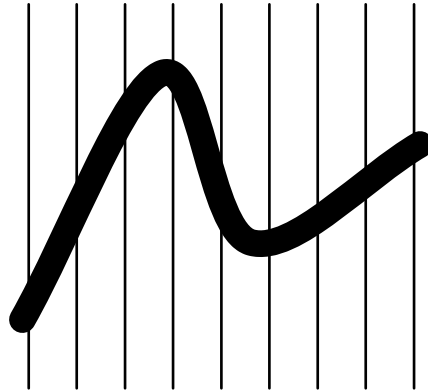


In practice, it is common to use one byte each for red, green, and blue, for a total of 24 bits—enough to describe 16,777,216 different colors, more than the human eye is capable of distinguishing.

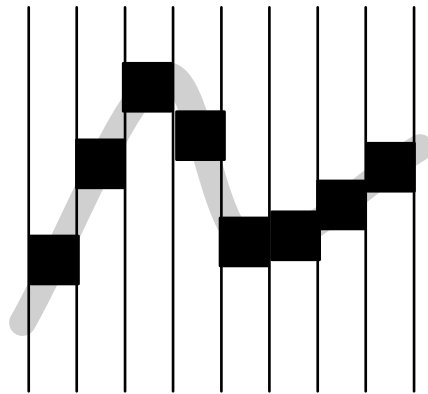
One more example: sound. A sound wave is a vibration in the air. So consider a (simplified) wave:



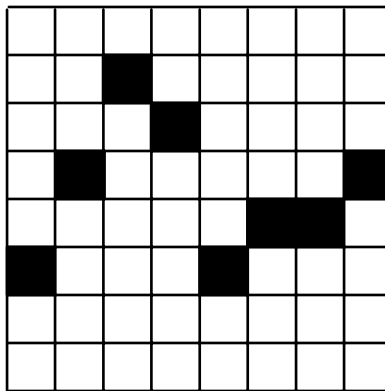
First, we can slice the sound wave up by time, into a series of discrete intervals. The most common ways of representing sounds in computers uses 44,100 slices, or *samples*, per second, since this was the frequency used for CDs. Imagine that each of these slices is 1/44,100 of a second long:



Now, within each sample, we need to say how loud the sound wave is at that instant in time. We'll take the *average* loudness in that  $1/44,100$  of a second:



Finally, we round off, or *quantize*, the height of the sound wave within each sample the same way we rounded off the shape of the face within each pixel and the intensity of each color:



The first sample has a loudness—a height—of 3 units, the second a loudness of 5, and so we can represent these eight samples as the sequence of numbers 3 5 7 6 3 4 4 5. These numbers can, of course, be turned into bits by writing them out in binary, just like we did above.

We have now seen how to *digitize* words, images, and sounds: to represent them using bits in a digital computer. The same techniques work for other media. A video, for example, is just a sequence of images together with an accompanying soundtrack—but we already know how to encode both images and sounds.

### Hardware

Some computers look familiar: they have a keyboard, screen, headphone jack, and so on. Other computers, like the one inside a pacemaker or a car engine, look radically different. But they all share a common design. Every useful computer contains at least two kinds of physical components, or *hardware*: a *central processing unit*, or *CPU*, that is capable of carrying out various computations, like addition and division, and *memory*, which stores data for use by the CPU.

Think of a memory chip as a large wall of storage lockers, each of which can hold some bits. To store bits in memory, the CPU needs to pick a locker to put them in; to get the bits out again for later use, the CPU needs to remember which locker it used. A very simple memory chip, for example, might have 256 locations, each of which can hold one byte. Each location has an identifying number, like the number on the front of each storage locker. So if we stored the digitized image of a face from above in memory, it would look something like this:

Location	Value
0	11111111
1	11111111
2	10011001
3	10011001
4	11111111
5	10111101
6	10000001
7	11111111
...	

Here, the face is stored in locations 0 to 7 (for technical reasons, it is more convenient for computers and programmers to count starting at zero).

It should be no surprise that these addresses can also be encoded using bits. They're just numbers, after all, and we already know how to encode numbers. So each location in memory has an address encoded in binary, from 0 (00000000) up through 255 (11111111). Here's the same depiction of the memory containing the face, this time with the addresses in memory given in binary, the way the computer itself would refer to them.

Location	Value
00000000	11111111

Location	Value
00000001	11111111
00000011	10011001
00000100	10011001
00000101	11111111
00000110	10111101
00000111	10000001
00001000	11111111
00001001	00111100
00001010	10010000
...	...

So to get the first byte of the face out of memory, the CPU asks for the byte stored at location 00000000 (binary for 0). To get the last byte of the face, the computer asks for the byte stored at 00000111 (binary for 7).

Computers have used a stunning variety of technologies to store data. Hard drives magnetize small portions of a spinning disc; USB flash drives store tiny electric charges. Older computers stored data as glowing spots on TV screens, as marks on a paper tape, and even as sound pulses traveling through liquid mercury. Indeed, any technology that can create either of two different physical states and then reliably tell them apart will work as memory.

There is a difference in common usage—if not always at the level of technical detail—between short-term *memory* and longer-term *storage*. The former is faster but loses track of the information it is storing when the power goes off; the latter is slower but capable of storing information for much longer. Most computers that you are familiar with have both: so your laptop has RAM (“random access *memory*”) to hold the programs and documents you are working with, and a hard drive for storing them when you close the lid.

So much for memory. Now for the CPU. The CPU’s job is simple. A program consists of a list of instructions—we will see how in a moment—so the CPU takes the first instruction and carries it out, then takes the next instruction and carries it out, then takes the next instruction and carries it out, and so on forever. These instructions tell the CPU what to do with the data. For example, consider addition: an instruction might instruct the CPU to add 3 and 9.

That may not sound particularly useful, because we already know the answer is 12. Rather, a more useful program would be capable of adding different numbers, not just 3 and 9. So instead of directly referring to two numbers, an ADD instruction could refer to two locations in memory—say 204 and 180—and tell the CPU to add their contents. The CPU would ask the memory to tell it what number is stored in location 204 and what number is stored in location 180, and then add *those* numbers together. To keep track of



the result for the next step of the program, the CPU will then usually store the answer back in memory, at yet another location specified by the instruction.

The memory and CPU are the heart of a computer. Almost everything else is a *peripheral*: something attached to the computer so that it can receive data from the outside world or do something useful. A keyboard is a peripheral for typing text in: when you push a key, the keyboard sends a signal to the CPU telling it which key you pressed. A screen is a peripheral for displaying information: the CPU (typically with the aid of a sidekick processor dedicated to graphics) sends a signal to the screen telling it which pixels to turn on and how bright to make them. A fingerprint scanner is an input device that receives one kind of information; a webcam is an input device that receives a different kind. Indeed, from this perspective, even an Internet connection is just another kind of peripheral: it's a device that the computer can send information to and receive information from. The computer doesn't know what lies beyond—only that some bits went out and other bits came in.

### Software and Object Code

Just as bits can be used to represent numbers and characters, they can also be used to represent *instructions*: the smallest individual operations that a computer can carry out. On the chip used in the PlayStation 2, for example, the bits 100000 represent the “ADD” instruction that adds two numbers together, and the bits 011010 represent the “DIV” instruction that divides two numbers. Put a sequence of these instructions together and you have a program in *object code* (sometimes called “machine language”), that is, a program written in a format that the computer can recognize and act on.

In other words, a program is also a form of data. After all, each instruction in object code is made up of bits.\* Since we know how to store bits in memory, this means we can also store computer programs in memory. Indeed, this is exactly what modern computers do. The CPU doesn't just retrieve from memory the data the program works on. It also retrieves from memory the program itself, one instruction at a time.†

The point that programs are both the instructions that tell a computer what to do and a kind of data that can be stored in a computer was first articulated by the British mathematician Alan Turing in 1936. His insight is responsible for the fact that modern computers are *general-purpose* devices, capable of carrying out all kinds of tasks, including ones their designers never dreamed of. To put the computer to a new use, just write a program, load that program into the computer's memory, and tell the CPU to get to work running it. Instead of needing to build a different computer for each possible use—one for playing *Tetris*, one for writing emails, one for calculating averages, and so on—it suffices to build a single computer.

---

\* So, for that matter, is a program written in source code. The difference is that the bits in source code represent letters and symbols for a program that will have to be compiled into object code, rather than representing instructions the CPU is directly able to carry out.

† If the CPU had an interior monologue, it might go something like this:

Hey memory, can you look up an instruction for me in location 480? 100000, got it, thanks! Let me see, 100000 means add, so I need to add two numbers. Hey, memory, can you look up some data for me in locations 204 and 180? 3 and 9, got it, thanks! Let me see, 3 plus 9 equals 12. Hey, memory, can you put the number 12 in location 184? Thanks!

Hey memory, can you look up an instruction for me in location 484? 011010, got it, thanks! Let me see, 011010 means divide ...

Put another way, provided the computer's hardware is sufficiently powerful (and Turing showed that even an extraordinarily simple computer is good enough), all of its "smarts" come from the programs, or *software*. By loading a different program, or *application*, into a computer, we can make it play *Tetris*, or write emails, or calculate averages, or anything else we can express precisely enough to put into a computer program. As in the legend of the golem, a computer is an inert lump of matter which is animated by the words we put into it.

### Programming Languages and Source Code

If she wanted to, a programmer could write object code directly, crafting a series of bits to describe the operations she wants the computer to carry out. In the early days of computing, this *was* programming. Unsurprisingly, it was difficult, tedious, and error-prone. In response, computer scientists developed *programming languages*, formal artificial languages for writing programs. These languages are much closer to natural human languages, like English and Wolof, in that they are written using familiar alphabets and symbols, rather than ones and zeros, and are designed to be easier for humans to read. But they are like object code in that they are intended to express each idea completely unambiguously, so that each program does exactly one, completely predictable thing.

As an example, consider the process of averaging two numbers. If asked to describe averaging, you might say "Add the numbers together, and then take half of the result." This is an *algorithm*, a step-by-step process for carrying out a calculation. Your informal plain-language description of it makes perfect sense to English speakers, but it's just of a bunch of gibberish to a computer. If you turn on a computer and type "Add the numbers together, and then take half of the result," nothing useful will happen.

A computer programmer's job consists of translating informal descriptions like this one into a sufficiently precise series of statements that a computer could execute them. Here's what she might write if she were using the popular Python language:

```
def average (x y):  
    sum = x + y;  
    return sum / 2;
```

The first line of this brief program says what it does: defines ("def") a function named `average` which computes something based on two numbers named `x` and `y`. The second line corresponds to the first half of our informal description: it adds ("+" ) up `x` and `y`, then stores the result ("=") in another number named `sum`. The third line corresponds to the second half of our informal description: it takes half of `sum` by dividing ("/") it by 2, then announces ("return") that this value is the answer we're looking for. The point of expressing it this way is that each individual step, like + and /, corresponds to something so simple that the computer is already capable of carrying it out. A program assembles these elementary steps, one by one, into a more complicated, and usually more interesting, whole.

Having defined `average` in terms of simpler steps like + and /, the programmer is now free to treat it as a simpler step when defining other functions, like a homebuilder who puts floorboards down on top of the beams she put in place yesterday. She could write `average(2,4) + average(10,20)` and the computer would correctly answer 18. Now, `average` is just one function, while modern operating systems like Windows contain millions of functions, but the basic principle of using functions to build other functions is the same.

There are thousands of programming languages, which are useful for different purposes. The same algorithm may look quite different when written out in different languages. Here is a version of average in the commonly-used C language:

```
int average (int x, int y)
{
    int sum;
    sum = x + y;
    return sum / 2;
}
```

The details are slightly different, but this should be recognizable as a close relative of the Python version. On the other hand, here is a version of average written in the less-commonly-used Scheme language:

```
(define average
  (lambda (x y)
    ( / (+ x y) 2)))
```

This version may look extraordinarily different from the previous two, but does the same thing: averages two numbers. Some programmers—although probably a distinct minority—would even consider the Scheme version simpler and more elegant than the Python and C versions.

A program written in a human-readable programming language like Python is called *source code* to distinguish it from the object code that a computer runs. But this raises a complication. Source code is closer to being suitable for a computer to carry out than an English description of a program would be, because source code is formal and precise. But there is still a difference. “def average (x y)” is an arbitrary string of characters as far as the computer is concerned, just like “Add the numbers together.” The computer is looking for 100000, not “def” or “add”.

Thus, when a programmer writes source code in a human-intelligible programming language like Python, C, or Scheme, she must then transform this source code into something that the computer can act on. The most common way to do this is for her to write a special-purpose program, called a *compiler*, that reads source code and translates it into object code. This translation is possible because programming languages are precise enough that the meaning of a program is fully specified.\*

### Operating Systems

In theory, one could write a program that takes complete control of a computer and tells it everything to do. This is how old-school 8-bit Nintendo cartridges work: each game contains a complete program that is responsible for every pixel on the screen. But this is incredibly cumbersome for more complicated programs. If you want to write a program to keep track of your knitting patterns, for example, the last thing you want to do is write software to recognize individual keypresses or for drawing the title bar at the top of the window. And if each program stands completely alone, the only way to switch from one to another is to turn the computer off and on again—not exactly convenient when you want to look up a citation in one window and then type it into a brief in another.

Thus, modern computers run an *operating system*: a program that takes care of the administrative details so that the *applications*—programs to carry out useful tasks the

---

\* For a brainteaser, ask yourself how it’s possible to write a compiler. Isn’t the source code for the compiler useless unless the compiler already works?

user cares about—can focus on their particular jobs. Popular operating systems include Apple OS X, Microsoft Windows, and Linux. Cell phones are computers, too: their operating systems include Apple iOS and Google’s Android. An operating system starts running when the computer turns on and never stops. Even when an application is running, the operating system is waiting in the wings, ready to step in if the application needs help or tries to misbehave. Typically, an operating system offers its services to applications through a set of *application programming interfaces* (or “APIs”): functions that an application can call on if it wants to carry out some specific task, such as creating a window, playing a sound, or drawing a line to the screen.

Operating systems are so ubiquitous that it’s easy to overlook just how many problems they solve. A typical operating system does all of the following:

- *Hardware independence*: Computers come in all kinds of configurations and work with all kinds of hardware: they have different hard drives, CPUs, screens, graphics cards, game controllers, cameras, and network connectors, to name just a few. The operating system frees applications from having to worry about the precise details of the hardware, making it possible to write an application that will run on hundreds of different devices.
- *Powerful features*: If you are writing a quiz game for the iPhone and would like to add a phone-a-friend feature, you don’t need to write from scratch the software to make a phone call. Instead, you can just send an `openURL` message with the telephone number to dial to a `UIApplication` object. What might have taken tens of thousands of lines of source code if you had to write them yourself can be done in two, thanks to the APIs that iOS makes available to applications.
- *Consistent look and feel*: Microsoft Windows looks different than Apple OS X does. That’s because the Windows APIs draw windows and menus in a different way than the OS X APIs do. Every program on Windows will be displayed in a similar way just because it runs on Windows and uses the Windows APIs. This consistency helps users orient themselves.
- *Multitasking*: Once you have an operating system to play traffic cop, you can run multiple programs at the same time. The point of a “windows” metaphor, or a list of “open apps,” is that each program is running, independently, and the user can switch between them at will. When a new message arrives, the operating system routes it to your chat program, not your word processor.
- *Multiple users*: An even more sophisticated version of multitasking involves letting more than one person use the same computer. Your personal computer rarely does this, but you interact regularly with computers that do. A web server, for example, is typically designed to interact with many different users simultaneously: Gmail wouldn’t work well if only one person could use it at a time.
- *Security*: Because computers contain all kinds of sensitive and valuable data, from love emails to tax returns, it’s important to keep them secure. The operating system typically plays a significant role in keeping data private and in protecting data from accidental or deliberate destruction. When you connect your computer to the WiFi network in a coffee shop, the operating system prevents other people from installing their own software on your computer or from snooping through your hard drive.

Jonathan Zittrain, whose book *The Future of the Internet* is excerpted below, has compared this typical computer design to an “hourglass.” The operating system sits at the narrow neck. Above it is a wild profusion of applications doing all kinds of jobs. Beneath it is a wild profusion of hardware with all kinds of technical details. The operating system

provides a standard layer that ensures a common experience for all those applications on all that different hardware.

### The Internet

You may have heard of the metaphor of the Internet as a “cloud”: big and opaque. In this section, we will systematically look inside the cloud to see how things work. What we will find may be less complex than you may have feared.

#### Networks and Protocols

Computer networks come in all shapes and sizes. There are networks between computers in the same room; there is a network that connects the International Space Station to earth. There are computer networks for cell phones, networks for playing video from your computer on your TV, even networks that connect a wireless mouse to your computer.

The key to every single one of these networks is the idea of a *protocol*: a specification that describes how computers should use the network to communicate. You can think of a computer protocol as being like a diplomatic protocol: when two delegations meet, there is a precise order of formal greetings, handshakes, and statements. It may look bafflingly formal to an outsider, but the diplomats use it to communicate important information to each other about their countries’ respective concerns.

Similarly, when two computers communicate, the protocol specifies every aspect of the technical process. A simple communications protocol along a cable might say, for example, that a message of binary 1s and 0s should be encoded as a series of electrical pulses of 500 nanoseconds each, with a 1 being a pulse at 1.5 volts and a 0 being a pulse at 0 volts. The sending computer turns the 1s and 0s into an electrical signal on the cable; the receiving computer looks at the voltage on the cable and turns it back into the 1s and 0s.

The enormous diversity of computer networks is possible because for each physical medium, there are different protocols designed to take advantage of that medium’s characteristics. The idea is similar to the way that different kinds of roads have different traffic rules. You can drive faster on a highway than in a parking lot; you can drive different kinds of vehicles on a city street than on a bicycle path; you drive on the right side of the road in some countries and the left side in others.

It is common to call a physical medium connecting two computers together with an appropriate protocol a *network link*. Here are some common (and less common) network links:

- Ethernet is a widely used protocol for local-area networking (e.g., within a building, rather than cross-country). Its physical medium is most often “category 5 cable,” a set of plastic-wrapped wires with a phone-like plug at each end. The Ethernet protocol specifies how computers connected by an Ethernet cable should “talk” by turning the information they want to send to each other into electrical pulses, how quickly they can talk, and what to do if two of them start talking at the same time.
- Many computers use WiFi for their local-area networks. Here, the physical medium is electromagnetic radiation, i.e. photons zipping through the air at the speed of light. Each computer using WiFi has a small radio transmitter/receiver. The WiFi protocol tells the radio transmitter on what frequencies to broadcast and listen, how

loudly and for how long to transmit, and what to do if someone else starts transmitting at the same time.\*

- Your cell phone also contains a radio, as do cell phone towers. Again, the physical medium is electromagnetic radiation. Instead of using WiFi frequencies and transmission rules, however, the phones and towers use protocols with names like EDGE, EVDO, and UMTS to specify how they should transmit information to each other.

- Internet signals can be carried over traditional copper or modern fiber-optic phone cables; the DSL and GPON standards, respectively, provide protocols for doing so. Cable companies use DOCSIS to do the same over cable connections. If you remember dialup, it used the PPP protocol to provide Internet access by having your computer make a phone call to a local phone number, and encoded the data transmissions as audio (which is why picking up another extension and making noise would generally destroy the connection).

- Fiber-optic “backbones” provide long-distance connections on land and via undersea cable. They are engineered for super-high transmission speeds, using highly specialized protocols.

- Computer data can even be transmitted via carrier pigeon. Here, the pigeon is the physical layer, and the protocol specifies that data should be transmitted by writing digits on a piece of paper wrapped around the pigeon’s leg and secured with duct tape.†

### Inter-Networking and the Internet Protocol

The next complication is that not every computer is on the same small local network. Your computer has a direct network connection to only one or a few others. The overwhelming majority of computers in the world do not have direct connections to each other, and it would obviously be impossible to try. How do we use the diverse networks we have in order to build up something like the Internet, where it is possible for computers around the world to communicate? This is the problem of inter-networking, and the answer lies in something called the Internet Protocol, or IP.

The first key idea of IP is *routing*. Suppose that you want to download an MP3 from Amazon’s MP3 store. There isn’t a wire that directly connects your computer to Amazon’s computer. Instead, the information making up the MP3 is passed along from one computer to another – computers that are directly connected (by a wire or other network link) – until it reaches you. In essence, Amazon’s computer hands off the MP3 to a computer that is connected to it and is slightly closer to you. That second computer hands off the MP3 to a third, which hands it off to a fourth, and so on until it is handed off to a computer that is directly connected to yours, which hands it off to you. Computers that participate in the process are generally called *routers*.

Each handoff is, in essence, a computer-to-computer copy. The computer making the handoff transmits a complete copy of the data in the file to the next one. As soon as the receiving computer acknowledges that it has received all the data, the sending computer knows that it can delete its own copy. Transmitting information through the Internet thus requires making as many transient intermediate copies as there are computers in the chain from the original sender to the final recipient.

---

\* Thus, a standard wireless router is a device that has both a WiFi-compatible radio and an Ethernet-compatible jack. It translates messages that come in along the Ethernet link into WiFi radio signals, and vice-versa.

† No kidding. See D. Waitzman, *A Standard for the Transmission of IP Datagrams on Avian Carriers* (1990) (RFC 1149), <http://tools.ietf.org/html/rfc1149>.

Along the way, the data will travel over many different kinds of network links. It might start out on Ethernet inside Amazon's data center, then be transmitted along backbone links until it reaches your local area, then travel on a fiber-optic cable supplied by your phone company, and finally reach your computer via WiFi inside your home. All of these network links have one thing in common: they can be used to carry IP messages.

This is a truly profound idea. Network engineers would say that IP is *layered* on top of these various network links. The goal of any of the lower-level link protocols listed above is to create a network that is capable of carrying IP messages. Once that is accomplished, the IP message can be carried from computer to computer across multiple different networks: Ethernet, backbone, WiFi, etc. The message itself does not change in any significant way, even though the different link protocols will encode it in radically different ways on different networks.

This is why IP is called the *Inter*-net Protocol. It is designed to enable *inter*-networking: the tying together of different networks. IP plays the crucial role of giving these diverse networks a single common technical language. Indeed, this is where the *Inter*-net gets its name: it was an experiment in inter-networking that was so wildly successful that it became “the” Internet rather than just “an” Internet.

#### Routing and Addressing

But how do the computers along the chain know where to send the data? They may only be connected to a few other computers, but the data could potentially be going to any of the billions of computers on the Internet. How do they decide which of their neighbors to pass the data along to?

The answer is that each computer on the Internet has a unique *address*, called an “IP address” (named after IP, of course). An IP address is a 32-digit binary number; by convention, they are written as four decimal numbers separated by periods. For example, here are the IP addresses of a few well-known computers:

- apple.com: 17.172.224.47
- google.com: 74.125.91.99
- nytimes.com: 199.239.136.200
- mit.edu 18.9.22.69

Every message is carried in the electronic equivalent of an envelope with the IP address of its destination stamped on the outside. When a router receives a message, it examines the IP address on the message. If that IP address is the router's own address, then the message has reached its destination and the process is done. Otherwise, the router examines a large database called a “routing table,” which tells the router, what the next intermediate destination should be for any possible ultimate destination. For example, a router's routing table might say that all messages for google.com and apple.com (which are on the West Coast) should be given next to its neighbor A, but that messages for nytimes.com and mit.edu (which are on the East Coast) should be given to its neighbor B.

Each router has its own routing table. The process of constructing them is one of the most complicated and intricate aspects of keeping the Internet functioning. At a high level of generality, what happens is that individual routers tell each other what computers they know how to get messages to. The information gradually propagates throughout the Internet, until – in theory – every computer knows how to reach every other computer.

### Packet-Switching

The next complication is that most messages are too big to send all at once in this fashion. Instead, they are broken down into smaller *packets* (sometimes also called “datagrams”). Each packet is sent separately, like a jigsaw puzzle that is broken down into individual pieces, each of which is sent in a separate envelope to the same destination. Along the way, they may travel by different routes, depending on factors like temporary congestion in some parts of the Internet, or routers coming on- or off-line and thus becoming available or unavailable to pass packets along.

Packet switching may seem counterintuitive, but it has some notable advantages. One is that it is much more efficient than the alternative of “circuit-switching,” i.e., holding a dedicated connection all the way from sender to recipient open for the entire duration of the transmission. Circuit-switching commits to a single chain of computers from source to destination, but packet-switching allows the transmission to respond to moment-to-moment changes in the Internet, taking advantage of faster routes and avoiding sudden traffic jams. Packet-switching also avoids tying up the intermediate computers when there is no data flowing; think of a streaming concert video, where the flow of information will last for hours, but is much less than the full capacity of any of the routers along the way. In addition, as we will see shortly, packet-switching can be very resilient to errors.

These three big ideas – routing, addressing, and packet-switching – collectively characterize the Internet Protocol. As its name suggests, IP is central to how the Internet works. Indeed, “the global network in which computers communicate using IP” comes very close to being *the* technical definition of the Internet. We will see throughout the this book how these technical features have important consequences for the law.

### Reliable Transport

IP is not the only protocol that matters on the Internet. Instead, network engineers commonly speak of a *protocol stack* of multiple protocols in use at one time. The “stack” metaphor captures the idea that these protocols are *layered*: ones at higher levels take advantage of the services offered by the ones at lower levels to do their jobs. Here is a simplified view of the protocol stack used by a typical home computer:

- Application (e.g. email, web, etc.)
- Transport (TCP)
- Network (IP)
- Link (Ethernet)
- Physical (category 5 cable)

We started off by discussing the physical and link layers. Then we saw how the network layer – IP – ties different networks together into a single Internet with world-wide addressing and routing. Now it is time to move up again.



The next layer above IP in the protocol stack is the transport layer, and the most common protocol there is TCP, the “Transmission Control Protocol.” It has several jobs, but the most significant is “reliable transport”: that is, making sure that every piece of a message reaches the destination. IP is a so-called “best efforts” protocol; routers will do their best to make sure that packets get where they should, but they make no promises. Bad stuff regularly happens that causes packets to be lost. Sometimes a router is congested, with too much incoming traffic, and it needs to start “dropping” packets in order to cope, like an overworked mail carrier tossing some envelopes in the river. At other times, transient conditions, like electrical interference or a bug in a router’s software, can cause a packet to be scrambled so badly that the data in it is unrecoverable.

TCP deals with all of these problems through good bookkeeping. The sender and the receiver each maintain a list of the individual packets making up a transmission. As the receiver receives each packet, it checks off that packet on its list and informs the sender that it has. If the receiver realizes that it is missing a packet – for example, because it is receiving more recent packets without having received an older one – it asks the sender to retransmit the missing packet. Meanwhile, the sender is keeping track of which packets the receiver has acknowledged. If too long a time passes without an acknowledgment from the receiver, the sender assumes that something has gone wrong and initiates retransmission on its own.

This is why packet-switching can be surprisingly more error-resistant than sending an entire message at once. It is true that, as with a jigsaw puzzle split among ten thousand envelopes, there are more ways for something to go wrong. But if a few packets go missing, TCP sees to it that just the missing ones are retransmitted, rather than needing to start the entire message from scratch. To continue the analogy, if a few puzzle pieces are missing, it’s easier to resend just the missing ones than to mail the entire puzzle again. Similarly, because the packets are smaller, they are less likely to suffer an error than a larger message would be. A single jigsaw piece can be mailed in an ordinary envelope; the entire assembled puzzle will require a special oversized padded mailer.

TCP is also responsible for “flow control”: the process of determining how fast the sender slings packets through the Internet toward the receiver. If you have a good fiber-optic connection, you would obviously prefer to send packets faster than if you are connecting through a slow satellite connection. Put another way, TCP automatically adapts on the fly to the amount of available bandwidth between sender and receiver. (The actual algorithms it uses to do so all involve clever communication between sender and receiver, and have been tuned over the years to values that seem to work well.)

Here, we can see another advantage of layering. TCP can completely ignore the details of the underlying network. It doesn’t need to know whether it’s running on a WiFi network or on Ethernet or whatever. It can delegate all of those details – along with the details of routing – to lower-layer protocols. TCP is only responsible for reliable transport and flow control, so it can focus on doing its job well. Unsurprisingly, this helps make TCP simpler than if it also had to do all of these other jobs. Computer programmers would say

---

\* TCP is not the only transport protocol. Not every application needs to ensure that every single packet is delivered. A live voice chat, for example, is better off letting the audio cut out for a fraction of a second than waiting for seconds for every last bit to arrive. Multiplayer video games often prefer to minimize transmission delay so that players can respond more quickly to each other. These and other applications often use their own, specialized transport protocols. They have in common with TCP and with each other that they all depend on IP: each of them uses IP to transmit its packets, they just do different things with the results.

that layering is a kind of “modularity”: separating out different functions into smaller pieces makes them easier to get right.

### Applications

At last we arrive at the part of the Internet you are probably most familiar with: applications. These are the programs that actually do things, like email, web browsing, and instant messaging. They use TCP/IP\* and other lower-level protocols to move data back and forth, and then do interesting things with it.

The first important detail here – one you are likely already familiar with – is the idea of *clients* and *servers*. A server is a computer that has a particular resource or that does a particular job. For example, the computer that stores your law school’s webpage is a server, unsurprisingly called a “web server.” Other common servers you probably use on a regular basis include email servers like Yahoo! Mail and your school’s email, e-commerce servers like the iTunes Music Store, and chat servers that tell you whether your friends are online.

A client is a computer that connects to a server to get information or have the server do something for it. If you look at your law school’s webpage, your computer is the client. It sends a message over the Internet to the web server, asking for the webpage; the server responds with a message that contains all the information making up the webpage. The process is similar with other servers. By convention, information that goes from a client to a server is *uploaded*; information that goes the other way, from server to client, is *downloaded*. When there is no clear distinction between which computer is the server and which is the client—and particularly when there are numerous computers acting both like clients and like servers—the relationship is said to be *peer-to-peer*.

Applications often have their own protocols, layered on top of TCP/IP and the other lower-level protocols. When one computer sends an email to another, it uses a protocol named SMTP to tell the receiving computer whom the message is from, whom it is for, what its subject is, and what its contents are. BitTorrent is a publicly published protocol for exchanging complete files. Skype uses a secret protocol to exchange voice messages. Games use their own protocols to update players’ computers on what everyone else is doing.

Like a computer, the Internet also has an hourglass architecture. This time, IP provides the narrow neck in the middle. Above it are millions of different applications. Beneath it are all kinds of different physical networks. Like an operating system, IP provides a standard middle layer that ties the different physical networks together into a common network capable of supporting any number of applications, even ones that no one has thought of yet.

### The Web

Perhaps the single most important application on the Internet today is the World Wide Web or “web.” The web actually consists of two closely related standards. The first is a protocol, the Hypertext Transfer Protocol (or “HTTP”), for sending webpages from servers to clients. The second is a format, the Hypertext Markup Language (or “HTML”) for encoding a rich experience with images, hyperlinks, and interactivity using nothing but raw text.

---

\* The two were designed simultaneously and are so frequently used together that they often go by this combined acronym.

Let us start by considering the process of obtaining a webpage from a server. Your web browser (e.g. Internet Explorer, Firefox, Chrome, or Safari) is a program designed to request web pages from servers and display the results. Suppose, for example, that you want to read the latest technology headlines from the New York Times, so you type “nytimes.com” into the the address bar of your browser. It uses TCP to send a message to the New York Times server at nytimes.com. In response, the New York Times server will send back a message containing the webpage itself. The rules of the road for this process – e.g., how the client describes the web page it wants, and how the server explains whether that web page is available or not – are governed by HTTP. If you have ever seen a webpage that displays the message “Error 404 not found,” then you have seen HTTP at work. 404 is the error code used by HTTP to signal that the webpage the client asked for does not exist.

What you have obtained from the server is not yet a webpage, only a long text file. You can examine the details by going to any webpage and selecting the “View Source” command in your browser.\* What you will see is a set of instructions for displaying the webpage you are looking at. This is the actual, literal data that was sent from the server to your computer; your web browser is then able to transform the data it into the webpage you see. (When people talk about “the source” of a webpage or “the HTML” for the page, this is what they are referring to.)

Try this, for example, at your favorite news site or blog. Pick a headline, and then try to find it in the page’s source. You should be able to pick it out, along with a lot of things between angle brackets, i.e. “<” and “>” called *tags*. These tags are the instructions, which your browser turns into visible formatting in the webpage it displays to you.† Here is some simple HTML:

```
<li>I agree. We <b>have</b> been here before, as the <a
href="http://nytimes.com">New York Times</a> recognizes.</li>
```

When displayed by your browser, this text will look more like this:

- I agree. We **have** been here before, as the New York Times recognizes.

What’s different between the source and the displayed version? First, the <li> tag, which stands for “list item,” tells your browser that what follows should be formatted as a bulleted item in a list. The matching </li> tag (which has a slash before the li) marks the end of the item. Second, the <b> tag tells your browser to format the following text as bold, up until the matching </b> tag marks the end of the boldface segment. And third, the <a> tag, or “anchor,” specifies that the following text is a hyperlink. If you click on it, your browser loads the web page it points at, in this case the New York Times’s homepage. How did your browser know which new webpage to load? It uses the location specified inside the <a> tag, following the “href”‡ – in this case, “<http://nytimes.com>”.

One last HTML tag is worth explaining: <img>. Here is an example:

---

\* In most browsers, this is available under the “View” menu.

† Not every tag has visible consequences. In Chapter 6, you will encounter “meta tags,” which carry information about the page (intended to be used by search engines), and which are not ordinarily shown to normal web users. You can inspect them, however, by using the View Source command.

‡ “href” is a less obvious abbreviation than some of the others; it is short for “hypertext reference.”

```
Yes, I've seen it, but I have no idea where they got the name  
from: 
```

This will turn into the following in a browser:

Yes, I've seen it, but I have no idea where they got the name from:



Here, the `<img>` tag tells the browser that it should display a particular image in that position. The image isn't sent as part of the webpage itself. Instead, your browser, when it sees an `<img>` tag, sends an additional request to the server with the image. (Here, that server is `james.grimmelman.net`, and note that the server where the image comes from need not be the same server as the one where the webpage came from.) The browser then drops the image into the place on the page where the `<img>` tag was. You can think of the tag as being a kind of placeholder for the image, one that includes instructions for how to fill in the place with a specific image.

### The Domain-Name System

Another application is especially important to the functioning of the Internet. The domain-name system converts human-readable names (like “google.com” and “icanhascheezburger.com”) into the IP addresses used by computers.

When you look up a domain name – say, `my.nyls.edu` – what really happens? The process works hierarchically, from right to left. Any URL, such as `http://my.nyls.edu/cp/home/loginf`, can be broken down into three parts. The `http://` at the start is a protocol identifier, which indicates that this is a request for a web page. The `my.nyls.edu` in the middle – everything up through the next slash – is the domain name that identifies the server from which you're requesting the web page. And the “`cp/home/loginf`” part (everything following the slash) identifies to the server which particular page you are asking for.

The general rule is that if your computer (e.g., your web browser, when you type a URL into the address bar) asks a domain-name server to look up a domain name, it will tell you the IP address of the computer with that domain name if the server knows. If the domain-name server doesn't know about that particular domain name, the server will give you the IP address of another domain-name server that can help you. That is, it will either help you or respond with the technical equivalent of “I don't know, but here's someone who might.” Here's a simplified example:

- (1) You start by asking the “root name server” what it knows about `my.nyls.edu`. The root name server “understands” the last part of the address, here `my.nyls.edu`. It tells you that another computer – the so-called “top-level domain (TLD) name server” for all “.edu” sites worldwide – can help, and gives you the IP address for the TLD name server.

(2) You ask the TLD name server for .edu what it knows about my.nyls.edu. This server “understands” the second part of the address, here my.**nyls**.edu. It tells you that another computer – the name server for nyls.edu – can help, and gives you the IP address of this other name server.

(3) You ask the name server for nyls.edu what it knows about my.nyls.edu. This server “understands” the third part of the address, here **my**.nyls.edu. It gives you the IP address for my.nyls.edu directly. Armed with the IP address, your computer can now directly contact my.nyls.edu.

This process could in theory be iterated repeatedly, although in practice it rarely continues for more than a few steps.

### Cryptography

Cryptography – the science of secret communications – deserves its own discussion. As a starting point, consider the problem facing Alice, who would like to send a message to Bob. So far, so good, but Alice is also worried about Eve, who may be able to intercept the letter in transit (perhaps by sneaking into the post office, or taking it from Bob’s mailbox before he can open it). So Alice would like a way to *encrypt* her message so that Bob can read (or *decrypt*) it but Eve cannot.

#### Traditional Cryptography

Here is an extremely simple form of encryption: rot13. In this code every letter is replaced by the letter thirteen letters later in the alphabet (the name is short for “rotate by 13”). Thus A becomes N, B becomes O, C becomes P, and so on. The full table is:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

Thus, suppose Alice wishes to encode the message (or *plaintext*) THIS CODE SUCKS. She turns T into G, H into U, and so on. When she is done, she has the coded message (or *ciphertext*) GUVF PBQR FHPXF. Eve stares at this gibberish, mystified. But Bob, who knows the secret, simply rotates every letter back by 13 places. G becomes T, U becomes H, and so on, until he has THIS CODE SUCKS again. Bingo! Secret secure.

The only problem is that rot13 is a terrible code, because it is so widely known. Any lengthy English text encoded using rot-13 will have many words starting with GU in the ciphertext (e.g. THIS becomes GUVF and THE becomes GUR). This is a dead giveaway that Alice and Bob are using rot13. Better codes reduce these predictable patterns.

Inventing a new code from scratch for every use would be a lot of work. Instead, cryptographers achieve better security by creating families of codes that use the same encryption algorithm but can have many different possible *secret keys*. Metaphorically, Alice puts the message in a box and seals the box with a lock that only the secret key can open. Bob, who has the key, can open the lock and read the message; Eve, who doesn’t have the key, is out of luck. If Bob wants to reply to Alice, he can use the secret key to lock the lock again; again, Alice can open the lock but Eve cannot. Even if Eve recognizes the general type of lock, it tells her nothing about which particular key opens it.

For example, a slightly more secure version of rot13 is rotN (“rotate by N”). Alice and Bob pick a number between 1 and 26 to serve as the secret key. To encrypt, Alice rotates each letter in the plaintext forward that many places in the alphabet; to decrypt, Bob rotates each letter back by the same number. rot13 is rotN with the secret key 13.

Unfortunately, rotN is still a terrible secret code. For one thing, it is vulnerable to a *brute force* attack. Eve can simply try rotating an encrypted message by one letter, then by

two, then three, etc. With the aid of a computer, she can try each of the 26 possible secret keys in a small fraction of a second. Better codes have more possible keys, so that brute force attacks take much longer. Fortunately, just as the number of possible values a computer can store grows exponentially with the number of bits, so too the number of possible keys grows exponentially with the number of bits in a key. A code using 256-bit keys is not twice as hard to brute-force as a code using 128-bit keys; it is  $2^{128}$  times harder.

Brute force can be surprisingly effective, particularly when it comes to that most familiar of keys, the password. There are 208,827,064,576 eight-letter all-caps passwords. If a would-be hacker types in each possible password, one after the other, at a rate of one password every five seconds, it will take about 33,000 *years* of nonstop work to try them all. But if the hacker runs a program that can try a million passwords per second, it will take only about two and a half days. Basically, password guessing is futile, but a determined brute-force attack will succeed against many short or simple passwords.

Another problem with rotN is that it is vulnerable to *cryptanalysis*. Because it is a simple substitution cipher, in which each letter in the plaintext is consistently replaced by the same letter in the ciphertext, encrypted messages replicate all of the patterns of English, e.g. some letters are much more frequent than others. If Eve has a sample of Alice's messages and notices that the letter G appears more often than any other, she may guess that it represents E, so that Alice and Bob are using the secret key 2. Better codes disguise these patterns, so that even small changes to the plaintext or the secret key create large and unpredictable differences in the ciphertext, and so that encrypted messages are close to indistinguishable from completely random gibberish.

One well-known state-of-the-art algorithm is the Advanced Encryption Standard, or AES. It encrypts messages 128 bits at a time, using a secret key of up to 256 bits. It mixes the bytes of the plaintext by repeatedly scrambling bytes, adding them together, and rearranging them. Although the process is long, and requires dozens of steps, every step is easy to reverse—for someone like Bob who has the secret key. Cryptographers currently believe that it is infeasible for someone like Eve who lacks the secret key and must guess at it to decrypt encoded messages. Cryptographers have thought that about other algorithms before, and been wrong; they may yet discover that AES is also vulnerable.

Even a stronger encryption algorithm like AES doesn't solve all of Alice's and Bob's problems. Eve could find a way to steal the secret key from Bob. Or Alice might leave a copy of the plaintext lying around. Or Eve could kidnap Bob and force him to turn over the key. No cryptographic algorithm can guard perfectly against these other attacks. Cryptography is just one component of the larger project of computer security.

### Public-Key Cryptography

This is essentially where the state of the art in cryptography stood as of 1970: Alice and Bob needed to share a single secret key, used for both encryption and decryption. In the next decade, though, two teams of researchers turned the world of cryptography on its head.\* In 1976, Whitfield Diffie and Martin Hellman, drawing on work by Ralph Merkle, published the idea of *public-key* cryptography. Their crucial idea was that only the decryption key held by Bob really needs to be secret; if the encryption and decryption keys can be separated, then the encryption key can be public, and anyone from Alice to Zelda can use it to encrypt messages to Bob. As long as Bob keeps his *private key* secret, he can pub-

---

\* A third team, working inside the United Kingdom's Government Communications Headquarters (GCHQ), hit upon many of the same ideas independently and earlier, but since spy agencies don't publish, their work remained unknown to the world for decades.

lish the *public key* to the world, including Eve. In the metaphor, the public key is like a lock blueprint: anyone can use it to manufacture locks that only Bob can open. Whenever Alice needs to send Bob a message, she manufactures a Bob-only lock.

As described, the Diffie-Hellman scheme was only an idea, not a workable system. But the next year, in 1977, Ron Rivest, Adi Shamir, and Leonard Adelman published an actual algorithm (now known as RSA after its inventors) that made public-key cryptography a reality. The insight behind the RSA algorithm is that as far as we know it is much easier to multiply numbers than to factor them. To simplify slightly, to create a secret key, Bob picks a pair of very large prime numbers  $p$  and  $q$  at random. He then uses their product,  $n = p \times q$ , as his public key. To encrypt a message, anyone can write out the plaintext as a number, raise that number to a high power, and take the remainder after dividing by the public key  $n$ . Thanks to some elegant mathematics, it is easy to undo the process but only if you know what the factors of  $n$  are. Bob knows that  $n = p \times q$ , but anyone else cannot decrypt the message without factoring  $n$ , which mathematicians believe to be impractically hard. Since the RSA breakthrough, numerous other cryptographic algorithms have been built up from the same algebraic building blocks: multiplication, powers, and remainders.

The asymmetry of public-key encryption opens up all kinds of new and interesting possibilities beyond simply keeping messages secret. For example, in many systems, Bob can use his private key to “sign” messages: only someone with the private key can generate this *digital signature*, but anyone with the public key can check that whoever signed it really did know the private key. Metaphorically, the digital signature lets Bob use his private key as a stamp, leaving an unforgeable imprint on the documents he signs. Bob could even combine a signature with encryption: signing a message with his private key and encrypting it with Alice’s public key. Now Alice knows that the message is genuine and she is the only person who can read it.

Digital signatures and related technologies have many, many applications:

- If Bob is careful about guarding his private key, he can use his digital signature as a source of *authentication*: anyone receiving a signed document knows that it was generated by Bob. He could tell his correspondents not to trust anything purporting to come from him unless it is digitally signed. Banks and other financial institutions use signatures in this way to prove the authenticity of transactions.
- Digital signatures can also be used to certify the *integrity* of a document. If anyone unauthorized tries to alter the document, the signature will no longer correspond to the altered document. Almost anything can be signed this way; some states now use digital signatures to establish the authenticity and integrity of their judicial opinions.
- Sometimes bits can be scrambled during the transmission of a message by cosmic rays, power surges, or bugs. A receiver who checks the signature against the message can use it as a form of *error detection* and see that something has gone wrong and ask the sender to transmit it again. Hard drives and other storage systems use error-detection algorithms to guard against accidental data loss. If data in storage is corrupted, the error can be spotted before it spreads, and the bad data can be replaced with a clean copy known to be good.
- It is often convenient to use short digests of documents as a shorthand for talking about them. So, for example, a two-megabyte image could be converted to a 256-bit *hash value*.<sup>\*</sup> Two images with different hash values are definitely different; two images

---

<sup>\*</sup> Not to be confused with a Twitter hash *tag*.

with the same hash value are extremely, extremely likely to be identical. The hash values provide a fast way to check whether two images are the same—very useful if you run a site where users upload images and you don’t want to store the same image again and again. Some hashing algorithms are designed to be secure: even an adversary who is determined to find two documents with the same hash value is highly unlikely to succeed.

- A *watermark* embeds one message in another with the goal of making it possible to tell where a given file came from. Stock photograph websites watermark their sample images; movie studios watermark the promotional copies they distribute to media outlets. Watermarks need not be visible to humans; ideally they should be hard to remove. In 2000, the Recording Industry Association of America threatened a lawsuit against a team of academic researchers who showed it was easy to remove the watermarks in a proposed scheme for watermarking audio files.

- In *steganography*, the goal is to hide one message undetectably inside another. A crude steganography scheme might use one space after a period to indicate a 0 and two spaces to indicate 1, hiding a short coded message inside an email. More sophisticated forms of steganography tweak small bits in an image or video: tiny variations in color or sound levels are completely imperceptible to humans.

One hard problem in any encryption system is *key distribution*: how do Alice and Bob learn each others’ keys? There is a chicken-and-egg problem here: unless Alice and Bob already have a secure way to communicate, Alice cannot be sure she is talking to Bob rather than to Eve pretending to be Bob. One common solution is for some trusted third party to certify Alice and Bob’s identities (using its own digital signature, of course): either a major hard-to-imitate institution like Google, or a mutual friend. So, for example, *certificate authorities* like Symantec or Comodo issue *certificates* using the X.509 standard: these certificates contain the public key for some entity and are signed with the certificate authority’s own secret key. Of course, certificate authorities have their own key-distribution problem: how do you know what the authority’s public key is? For this reason, browser makers frequently include certificates; anyone using Firefox already has a copy, supplied by Firefox, of Comodo’s public key.\*

#### An Example

Real-world computer systems frequently combine many cryptographic techniques together into more complicated protocols. For example, Transport Layer Security, or TLS, is a widely used protocol for clients and servers to communicate securely. Here is a simplified version of the process:

1. Well in advance, the server obtains an X.509 certificate that lists its domain name and RSA public key.
2. The client sends a simple “hello” message to the server indicating that it wishes to communicate securely.
3. The server responds with its own “hello” message that includes the server’s X.509 certificate.
4. The client checks the validity of the X.509 certificate. Among other things, it verifies:

---

\* If it occurred to you to ask how you know that you received a properly authenticated copy of Firefox when you downloaded it, rather than a modified version that includes a forged public key for an impostor pretending to be Comodo, congratulations, you are starting to think like an Internet security expert.



- That the signature is issued by a certificate authority the client trusts.
  - That the signature on the certificate is valid.
  - That the domain name listed on the certificate matches the domain name the client thinks it is contacting.
  - That the certificate has not expired or been revoked by the certification authority.\*
5. The client now knows the server's RSA public key. It sends the server an RSA-encrypted message containing a random number.
  6. The server uses its private RSA key to decrypt the random number. Both the client and the sever now know the random number, but no one else does. They can therefore use it as a shared AES key.
  7. The client encrypts its first substantive message to the server using their shared AES key and sends the encrypted message to the server.
  8. The server decrypts the message using the AES key, determines its response, encrypts that response using the AES key, and sends the encrypted response to the client.
  9. If desired, the client and server can continue the process using the same AES key as long as the connection stays open.

Among other things, TLS is the basis for HTTPS, the encrypted version of HTTP. When your browser shows you a lock in the address bar (or another visual indicator of a secure connection), it means that your HTTP messages to and from the server are encrypted using TLS, keeping the contents safe from eavesdroppers. The easiest way to request an HTTPS connection is to replace `http://` with `https://` when typing in a URL – although not all servers honor clients' requests to use HTTPS. Regular HTTP connections are unencrypted, so that anyone in a position to observe your messages (e.g. your ISP) can see what URLs you are browsing to and what those web pages contain.

Observe that in this example, TLS employs certificates (X.509), public-key encryption (RSA), and shared-key encryption (AES). This combination of smaller pieces is common in modern cryptography and enables some remarkable feats of security engineering. Unfortunately it also means that when one piece proves to be vulnerable, whole edifices are at risk. The infamous Heartbleed bug of 2014 was a coding mistake in the most common server implementation of TLS: by sending a particular kind of malformed message, a client could cause a server to send back the contents of its memory. Since a common use of TLS is logging into websites, that part of memory often contained users' passwords. Ironically, until the bug was fixed, changing one's password actually made things worse, because a user would typically then log in using the new password, exposing it to attackers using Heartbleed. Despite striking advances in cryptography, Internet security remains a hard problem.

### Internet Applications Problem

Familiarize yourself with the following:

- [Jason Kottke's blog](#)
- [Gmail](#)
- [Amazon.com](#)

---

\* Test yourself. What could go wrong if the client omits any of these checks?

- Skype
- World of Warcraft
- Twitter
- Facebook
- Google
- YouTube

You don't need to sign up for accounts or to use these applications, but you should be at least passingly familiar with them. They provide a useful range of examples. Do your best to answer the following questions for each of these applications:

- (1) What can you do using this application?
- (2) Does the application require that you and other users both be online at the same time? If so, how does the application figure out that you're both available?
- (3) How does the message get from your computer to someone else's (or vice-versa)? Is it stored anywhere along the way? Who could listen in or read it if they wanted?
- (4) How – in a very general sense – is the content encoded? Is it human-legible? Does its quality suffer in transit?
- (5) Are there servers somewhere that assist in making the application available? If so, do they store the content, or do they merely assist in making connections? Could you make connections without the assistance of a server? Who's in charge of keeping those servers running, providing them with electricity, and so on?
- (6) Do you need an account to post content? To receive it? How much information about yourself do you need to give up in order to participate?
- (7) Who's allowed to post content, and of what sort? Is this an egalitarian medium, or one in which only a few people speak and the vast majority only listen?
- (8) What happens "under the hood?" Is there a flow of information that you can describe in general terms, or does something so mysterious happen that it might as well be magic?