Handout 1

Data Representation

In this handout, we are going to provide some basic information about digital systems that you need to know to understand much of the rest of the material. If you have had prior courses in computer systems, some of the topics here will be familiar. If you haven't then it might not be, but don't worry. It is not rocket science, just computer science.

We are going to give you a glimpse of how computers work. We will show you what puts the "digital" in *digital systems* and how computers process information. We will also teach you a few principles of *cryptography*, which allows us to make sure that information is kept secret or unaltered. Finally, we will introduce some forensic tools so that you can use them in exercises, and on your own computer, as the course goes on.

1.1 It's All Just a Bunch of Bits

A computer is a tool that processes information. We know from experience that it can handle text documents, music, video, and a wide variety of other types of files. The way that the computer handles all these disparate formats is to store them all in a common encoding that the computer can work with efficiently. This encoding, which is the native encoding of all digital devices, is called **binary**.

In binary, everything is represented in ones and zeros, each of which is called a **bit**. Any number, any date, any program, any song or movie on the computer is a sequence of bits. It is just a bunch of zeros and ones in a row. This will hold true for data from any storage devices, like a hard drive or USB key as well. In fact, we can take all the data from those devices, containing all the files and other information as a single series of bits and save it as a single **disk image**. Later we can carve out individual **digital objects**, which are a smaller series of bits with a particular start and end and meaning. In other words, digital objects, such as files, are a subset of the bits from the series of bits making up the disk image.

The number of bits needed to store a piece of data increases with the amount of information encoded. Some things, such as value that is only true or false (often called a **boolean** value), don't take up much space at all. In theory, a boolean value can be stored in a single bit. Other things, like the video file of a movie, might take many billions of bits to store. All the storage space on a very large drive will hold billions or trillions of bits. *The point is that everything on the computer, from data to programs, is just a bunch of bits.* We will keep coming back to this point to remind you of it.

In performing forensic investigations, we will frequently be examining bit sequences that are left as evidence. It is important to realize that while recovering the actual bits is vital, we will also need to know the context of the bits to understand them. Are they part of an audio voice mail file? Do they represent a spreadsheet? Most often, the bits alone do not provide enough information to answer these questions — after all, they are just a bunch of bits. We will therefore need to recover other information necessary to determine the context for interpreting particular bits of information. This information can come from file names, or it can come from knowing how a particular set of data is structured. We will have many more examples of this throughout the course. First, we need to talk about how the computer handles bits, and how we can talk about them more easily. We will also examine a few of the different ways that data can be encoded into binary form.

Working with lots of bits

It is rare to work with individual bits. Instead, bits are most often bundled into groups and processed as part of that group. This is because a single bit has only two possible values, and we most often want to deal with objects that have many more values. It is also because the overhead of tracking many individual bits is far higher than tracking fewer groups of bits. Think about watching children. If you have a lot of them, it is easier to get them to hold hands to keep them together than letting them run around on their own.

There are many different sizes for the groups of bits. The most common is called a **byte**, and it is 8 bits together as a group. Though less common, half a byte is called a **nibble** and is four bits.

Also common are references to a **word**, which is the number of bits that a particular computer processor can handle at once. The size of a word is most often a power of 2. Most computers today use 16-, 32-, or 64-bit words, which is 2, 4, or 8 bytes. Since computers are optimized to work with a particular fixed size chunk of data, the word size is the smallest size group of bytes that a computer handle. All operations are conducted on a word-size chunks of bits.

The number of values a particular group of bits can represent grows exponentially as the number of bits grows. A single bit has two values; two bits can represent four different values; three, eight values. Most generally, the number of values that a group of n bits can contain is:

$$\text{number of values} = 2^n$$

n	2^n	Binary Representation
0	1	00000000001
1	2	00000000010
2	4	00000000100
3	8	00000001000
4	16	00000010000
5	32	00000100000
6	64	00001000000
7	128	00010000000
8	256	00100000000
9	512	01000000000
10	1024	10000000000

Table 1.1: Powers of Two and their Binary Representation

Because of this property, numbers that are a **power of two** are very, very common when talking about computers. Table 1.1 shows powers of two up to 2^{10} and their binary representation. The powers of two show up repeatedly in the sizes of different objects and storage devices, which are usually measured in bytes instead of bits. Because these numbers can get very large, it is common to add prefixes to the word byte (abbreviated B) to indicate the general size. The most common prefixes are: **kilo-**, which indicates a thousand; **mega-** which is a million; **giga-** for a billion; and **tera-** for a trillion. Each of these are abbreviated with their first letter, so 10GB is ten gigabytes, or ten billion bytes. Sometimes people instead measure bits and use the lower case b for this, as in 10Gb, or ten billion bits, which is only 1.25 gigabytes. This is most frequently seen when referring to how much data can be sent over a network connection.

Use of these prefixes can get somewhat confusing, though, because people use them to refer to both binary and **decimal** (or base 10) numbers, even though the values vary significantly. For example, in decimal numbers, 1K is a thousand, but the closest power of 2 is 2^{10}, which is 1024. Still, people often refer to each of these as 1K. This notation turns up most often in describing the size of hard drives. Manufacturers will advertise sizes in decimal numbers, but the operating system reports sizes in binary numbers. A drive that the manufacturer advertises as holding 500GB, or 500,000,000,000 bytes might be reported by an operating system as 465GB. Both are correct, but only for different interpretations of what G means — the first is decimal and the second is binary.

As a solution to this confusion, it is more precise to specify whether you are using decimal or binary prefixes. Binary prefixes are similar to decimal ones, but end in *bi-* (pronounced "bee") and include the letter 'i' in their abbreviation. For example, 1KiB is a **kibibyte**, or 1024. Other prefixes include **mebi-**, **gibi-**, and **tebi-**, which are abbreviated Mi, Gi, and Ti respectively. Table 1.2 shows some of the relations between these numbers, and how some of the sizes are commonly spoken.

n	2^n	Binary Prefix	Decimal Prefix	Spoken name
4	16			nibble
8	256			byte
	1,000		K	kilo
10	1,024	Ki		kay-bee
	1,000,000		M	mega
20	1,048,576	Mi		meb-bee
	1,000,000,000		G	giga
30	1,073,741,824	Gi		gib-bee
	1,000,000,000,000		T	tera
40	1,099,511,627,776	Ti		teh-bee

Table 1.2: Binary and Decimal prefix sizes

1.1.1 Representing Bits in Hexadecimal

In the practice of forensics, there will be times that we will want to communicate the contents of a bit sequence — the actual series of ones and zeros. We could try and read them or write them all out, but humans are not very good at dealing with lots of binary digits. We lose our place, miscount the number of digits in a row, or make other errors. To avoid this, we can use a more compact and error-resistant method of describing bit sequences called **hexadecimal** (or simply **hex** if you want to sound like a pro). In hex, we convert four-bit nibbles into digits and letters using a specific format. A hex encoding of a binary number is really a base-16 encoding, using sixteen different symbols: 0 through 9, and A through F. Table 1.3 shows how nibbles are converted to hex characters.

Using hex makes it much easier to work with bit sequences. For example, if we needed to communicate the bits

11110101101000101001011011101011

to another person, we could convert them into hex and then send the hex digits. To do so, first we split the sequence into nibbles and then replace each nibble with the appropriate hex character:

1111 → F
0101 → 5
1010 → A
0010 → 2
1001 → 9
0110 → 6
1110 → E
1011 → B

Then we combine the hex characters into a single group of characters to get

F5A296EB

which is far easier for people to communicate. You might have realized, though, that it is possible to still have confusion. Suppose that the bits you wanted to send were

Nibble	Hex Character
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Table 1.3: Hexadecimal Representation of nibbles

00010001000100010000000100000000

Converting that to hex, we get

11110100

which looks like it could be a binary string. To avoid this confusion it is common to add a marker to show that the string in in hex. There are two common ways to designate what encoding is in use. One is to add the base as a subscript to the string. For example, binary representation uses a subscript 2 as in 11110100_2; decimal uses a subscript 10 11110100_{10}; and hexadecimal uses a subscript 16, like this: 11110100_{16}. Even though all the digits are the same, they have different meanings. You will mostly see subscripts used in books, including some of our examples, since most software doesn't display subscripts well on the screen.

The other way of showing that a string is in hex is to add a 0x to the front of it. To be clear, that is a zero and not a capital letter O. This is more common, and is the convention we tend to use in this course, except where showing hexidecimal output of programs that do not do so. Formatted this way, the hex strings above appear as

0xF5A296EB

0x11110100

One thing that occasionally happens when converting from binary to hex is that you have fewer bits than evenly go into nibbles. For example, 101110_2 only has six binary digits, but would need eight to split evenly into nibbles. In these cases, just put enough zeros in front of the number to make it the number of binary digits divisible evenly by four, which is called **padding**. In this example, padding out the six binary digits gives us 00101110_2, or 0x2E.

Hex encodings are very common when we need to work with a bunch of bits. We will see them again and again when dealing with cryptographic keys and the output of hash functions Just remember that when you see that 0x in front, it looks like letters and numbers, but is really just a bunch of bits encoded in hex.

1.1.2 Binary Encodings

Everything in the computer is just a series of bits. What those bits mean, however, can vary greatly depending on what is encoded in them. The bits themselves may not provide any clues, so we will have to discover a context in which to understand them. Fortunately, since computers tend to be very rigidly structured and predictable in how they use bits, we will often have the context we need for interpretation.

Encoding Numbers

Integer numbers are one of the most common types of data that we will see in our forensics investigations. They show up again and again. Numbers are frequently stored by themselves, and many other types of data are encoded as a series of integers. For example, we will see shortly that computer dates and times are just integers, and images can be stored as many integers, each representing the color of a particular dot in the picture.

The simplest and most common encoding of integers into binary is just **base conversion**; for example, we might go from base 2 to base 10. Each digit in a decimal number represents a value from one to ten, and each place in the number represents ones, tens, hundreds or more, with each place ten more than the place prior. The value of the whole number is the sum of each place. So the number 623_{10} breaks down into:

$$6 * 10^2 + 2 * 10^1 + 3 * 10^0 = 623$$

where the superscripts represent the place from right to left. This is the same as saying:

$$6 * 100 + 2 * 10 + 3 * 1 = 623$$

A simple binary encoding works exactly the same way. Each digit represents one of two values, and each place from left to right represents the next higher power of two. Obviously since we have only two values and the powers of two go up more slowly than powers of ten, we will have more digits to represent the same number. The number 1001101111_2 breaks down into:

$$1 * 2^9 + 0 * 2^8 + 0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^1 + 1 * 2^1 + 1 * 2^0$$

which is

$$1 * 512 + 0 * 256 + 0 * 128 + 1 * 64 + 1 * 32 + 0 * 16 + 1 * 8 + 1 * 4 + 1 * 2 + 1 * 1$$

and since zero times anything is zero, and one times anything is one:

$$512 + 64 + 32 + 8 + 4 + 2 + 1 = 623$$

The binary representation is therefore 1001101111_2.

Converting to and from binary using a calculator. We want to teach you how things work, and not just how to use tools. So, as much as it pains us to say it, the easiest way to convert between binary and decimal numbers is to use a calculator. Since there

are times that a calculator might not be handy, and since we want you to understand what the calculator is doing, we will also show you how to do it yourself.

Most often you will have a calculator handy on your forensics workstation, and it can do binary, decimal and hexadecimal conversions for you.

- In Windows, the basic calculator can do both of these conversions. Depending on the version of Windows, the Calculator program is normally found under `All Programs` → `Accessories`. To do base conversion, select `Scientific` mode from the `View` menu. With the `Dec` button selected, enter whatever number you want converted to binary. When done, click the `Bin` button to see the binary representation. You can also select the `Hex` button to see what the hexadecimal representation of the binary number. To convert back, enter the binary data with the `Bin` button selected, then hit the `Dec` button.

- On a Mac OS X system, choose the calculator from `Applications` → `Calculator`, and under the `View` menu, choose `Programmer`. With the `Dec` button selected, enter the decimal number. The binary number will appear in the area above the keypad. To convert from binary to decimal, click on the individual digits in the binary area to set or unset them. The decimal conversion will appear above.

- Using Ubuntu Linux, open the calculator from `Applications` → `Accessories` and under the `View` menu, choose `Scientific`. With the `Dec` button selected, enter the decimal number, then choose `Bin` to see it as binary. You might also select `Bin` then enter a binary string for conversion.

- With Web access from any device, Google will do conversions for you as well. To convert from decimal, enter the query in the following format, changing the number to the one you want converted: `23 in binary`. The result will be given with a 0b in front of the binary string. To convert from binary to decimal, enter the binary string in the same format, with the leading 0b, with "to decimal", as in `0b01101011 to decimal`.

Doing Short Conversions by Hand or Mind Using a calculator can be helpful, but with a little practice, you can learn to do short conversions quickly either with or without a pencil and paper. The simple trick is to realize that any decimal number is just the sum of a number of powers of two. If you can work out which ones, then with practice it is easy to do the conversion.

Let's look at converting binary to decimal first. Suppose we have been given the bits 11001101, and we want to know the decimal value. Remember that each position on the binary string has a value as a power of two. Going from right to left, which is the reverse of how we usually read, the positions indicate the number of 1s, 2s, 4s, 8s, 16s, and so on. Each position has a value that is twice the last. Since we are working in binary, there can only be a one or zero in each position. To convert to decimal in your head, then, all you need to do is start at the rightmost position with one, and work to the left, adding up the values of the positions with a 1 in them. While this seems cumbersome, if you can multiply by two and add in your head, you can do conversions pretty quickly.

In our example, we have 11001101. Working from right to left, we see a 1 in the first position, so we start with 1 for our decimal value. The next position is twice the first, so we see that there are 0 twos, and our decimal value is still 1. Next is the 4 position, and we see a 1 there, so we add four to our decimal value, making it 5. Moving left again to the 8 position there is a one, so we add that to the total for a decimal value of 13. The next two positions for 16 and 32 are zeros, so they have no effect on the total. There is a 1 in the 64 position, so we add that to 13 to get 77. Finally, we reach the 1 in the 128 position and add that to get the final total of 205.

128	64	32	16	8	4	2	1
1	1	0	0	1	1	0	1

Working right to left:

$$\begin{array}{r} 1 \\ +4 \\ +8 \\ +64 \\ +128 \\ \hline 205 \end{array}$$

Now say that we want to convert the number 273 to binary. We know, either from experience or from Table 1.1 that 256 is the closest power of two that is smaller than 273. The 9th bit, representing the position of 2^8 (it is the 9th because we start counting with the first bit as 2^0), will therefore be a one. We subtract 256 from 273 and get 17. Repeating this, we see that 16 is the next largest power of two that fits. We now know that the 5th bit is a one, and subtract 16 from 17 to get 1. That is clearly equal to the lowest power of two, so the first bit is a one. We can just write out the binary number now, with ones where we discovered them and zeros elsewhere, giving us 100010001. This is not as fast as converting from binary to decimal, but once you know the powers of two it can also be pretty quick.

273	Largest power of two <= 273 is 256	100000000
$273 - 256 = 17$		
17	Largest power of two <= 17 is 16	000010000
$17 - 16 = 1$		
1	Largest power of two <= 11 is 1	000000001
	Combine the 1 bits into a single string	100010001

Negative Numbers In all our discussions so far, we have only mentioned positive numbers. We know, though, that we can and do use negative numbers. So how are those encoded?

The simplest way that you could encode negative numbers would be to add a single bit that represented the minus sign. It would be zero if the number was positive, and one if the number was negative. So, for example, positive 85 would be represented as 01010101, and negative 85 would be 11010101. In this case, the leftmost bit represents the minus sign.

Bit	Value	Binary Representation
0	1	00000001
1	2	00000010
2	4	00000100
3	8	00001000
4	16	00010000
5	32	00100000
6	64	01000000
7	-128	10000000

Table 1.4: Bit values for a two's complement byte

This type of encoding leads to a potentially odd result in that you could encode both a positive zero (000000000), or a negative zero(1000000). Of course, zero is neither positive nor negative so this represents an inefficiency that tends to annoy computer scientists. Modern computers tend to use a different encoding scheme that avoids this problem.

One such method is called **two's complement**. In two's complement, the values of the binary digits are changed so that the highest value bit is negative instead of positive. All other bit values are considered positive. Let's look at an example to make this clearer.

In a single byte, we have eight bits. These bits can represent $2^8 = 256$ different values. In the regular binary encoding, these values would range from 0 (00000000_2) to 255 (11111111_2). Notice that in the latter case the highest bit has a value of 128, and all the other bits sum to 127. Together they add to 255.

In two's complement, that largest bit is the negative of what the largest value would be in a regular byte, so the largest byte represents -128 instead of 128. Table 1.4 shows the values for each bit in a two's complement byte. The values of other bits are the same. The values representable in two's complement therefore range from -128 (10000000_2) to 127 (01111111_2). Decimal zero is still 00000000_2, but 11111111_2 is (-128 + 127) = -1. Any negative number will have the highest bit set, and will be negative, but we no longer have the problem with negative zero.

We have shown two's complement in a single byte here as an example. In computer systems, two's complement is implemented on the basis of words. On a machine that has 32-bit words (also called a 32-bit machine), the largest bit will have a value of -4,294,967,296, providing a range of -4,294,967,296 to 4,294,967,295.

Decimal Numbers We have seen how both positive and negative integer numbers can be represented, but we know that there is more to the world than nice, round numbers. We also have numbers that have a fractional or decimal part. We won't go into huge detail about the best ways to do this since it can be very complicated, but we do want you to understand the basic idea so you can understand why floating point calculations on computers can be wrong to some degree.

Bit position	Power of Two	Value
1	2^{-4}	$\frac{1}{16}$ (0.0625)
2	2^{-3}	$\frac{1}{8}$ (0.125)
3	2^{-2}	$\frac{1}{4}$ (0.25)
4	2^{-1}	$\frac{1}{2}$ (0.5)
5	2^{0}	1
6	2^{1}	2
7	2^{2}	4
8	2^{3}	8

Table 1.5: Bit values in a byte with a fractional portion

Imagine that we have a single byte, and we want to use it to represent a positive floating point number. One way we might do it is to split the byte into two nibbles, one representing the integer part and the other the fractional part. The integer nibble will represent $2^4 = 16$ different values, from 0 to 15. But what about the fractional part? For that nibble, we might assume that the bits represent 1 divided by the power of two moving away from the decimal. So the largest bit of that nibble is$\frac{1}{2}$, the next $\frac{1}{4}$, the second smallest $\frac{1}{8}$, and the smallest bit $\frac{1}{16}$. These are all powers of two, it is just that they are fractional powers of two. Table 1.5 shows what value each bit in this byte would have.

We know that to represent the integer part, we sum up the correct powers of two. For example, 10.00_{10} would be represented as the sum of 8 and 2, or in binary 10100000_2. Notice that there seems to be four extra zeros on the end of that number. Remember that in this example, we are working with floating point numbers, so the first nibble 1010_2 is the decimal part which adds to 10_{10}, and the decimal nibble is 0000_2, which is zero. If we wanted to represent a fractional number, we would choose the appropriate fractional powers of two. Say that instead of 10.00_{10}, we wanted to represent 10.75_{10}. The integer nibble would remain the same, 1010_2, but we would have to pick the correct bits to represent the .75. We would choose .5 and .25, so the fractional nibble would be 1100_2. The entire number would be represented as 10101100_2. The computer must keep track of what the bits represent in order to interpret this as a decimal number instead of an integer one, but we already know that happens because all bits also have context.

In our example above, 10.75_2 was easily converted into binary, as the .75 was an even sum of two of the fractional powers of two. Other numbers would be harder to represent accurately. Suppose we wanted to represent 10.35_{10} in binary with the encoding scheme we have been describing. Again the 10 part is easy, but we would have to select the fractional powers to two that added up to .35. Here we run into a problem. We can either use the sum of .25 and .125 = .375, or 0110_2 in binary, which isn't very close, or .25 + .0625 = .3125, or 0101_2, which is even worse. In this case, we would have to go with .375, and represent the entire number as 10100110_2. The difference between the

number we wanted, 10.35 and what we got, 10.375, is called the **representation error**, also referred to as **rounding error**, as the number must be rounded to the closest representation.

In our small example the representation error is large because we have only 4 bits to represent fractions. While we have been using a single byte as our example, most systems work with 32 or 64 bit words, so have much better accuracy because they use more bits to represent the fractional portion. For even more accuracy, some computers use two or even four words combined together to form on floating point number. This allow very high precision, and though they can still suffer from representation error at the extreme end of the fraction it is unlikely to be important.

A complete description of how decimal numbers are represented would be too much information for our purposes, though if you are curious you can look up the IEEE 754-1985 or IEEE 754-2008 standards. The important thing to remember, as always, is that decimal numbers are represented as a bunch of bits. The actual format of decimal bits varies, and as an investigator, you will need to know the context in order to understand them.

Encoding Text

If humans only used numbers to communicate, we would be done with our description of binary representation. But we know that people instead communicate with words and letters in a wide variety of languages. In order to be useful to people, digital systems need to represent the wide variety of written languages that people use. Fortunately, representation of characters turns out to be relatively easy in concept, though dealing with a multitude of different languages can be more complicated.

At the heart of all text representation is the tracking of individual **characters**, each of which is a symbol for a letter, number, or punctuation mark. Different languages use different symbols, so in order to avoid having to track all possible symbols at once, a variety of different **character encodings** are used. A particular encoding is really just a table of different symbols, each of which is referenced by a number. The letters or symbols in a file, then, are just a bunch of numbers, one for each character. The numbers are references to a symbol in the character encoding, which is part of the document. Since we have seen that computers are great with numbers, this is an efficient way to handle text. From a forensic standpoint, however, we have the added complication of needing to find enough context in order to determine the correct character encoding. Most forensic tools are able to do this for you, as files have the proper encoding embedded within or have a system-wide default.

Note that the encoding of a document is not the same as the font it uses. The font specifies how the symbols look on the screen or on paper. The encoding is what the symbols are used. You might have many different fonts in the same document, but you should only have one encoding.

There are a number of common text encodings that you are likely to find. One of the oldest and simplest encodings for English is called **ASCII**, which is short for American Standard Code for Information Interchange. ASCII was originally designed for use on

paper teleprinter machines. These machines did not have a screen and instead printed directly on to a long sheet of paper. ASCII encodes characters in 7 bits, leaving the 8th to allow for transmission error checking, if desired. (If you convert the hex values for each character to binary, you will see that the first bit is always zero). Since there are 7 bits, there are $2^7 = 128$ different characters encoded in ASCII. Not all of these are printable characters, however, meaning that they will not display in a readable way on a computer screen. They are instead special characters that control the action of the printer. Table 1.6 below shows the encodings of printable ASCII characters.

ASCII is very common, and formed the basis of most computer systems for many years. As you might guess, though, it is very limited in the languages that it can represent, since it contains only letters from English words and only has space for 127 characters total. Since computer and operating system manufacturers wanted to be able to sell systems to non-English speaking customers, other encodings were needed. As other languages have more letters and symbols, a larger number of bits was needed for these encoding.

A common eight-bit, or one byte, encoding is the ISO/IEC 8859 series of encodings. There are 15 separate parts, each different to a degree and targeted at a particular region of the world. This regional scheme takes advantage of the fact that similar languages use some common characters in their alphabets. A single encoding can therefore support multiple languages by containing a set of symbols that are common to all the supported languages, along with needed individual symbols for each separate language. In actual implementation, different systems companies have altered these standards to fit their own needs. For example, Microsoft instead uses their own encodings, which they call **code pages**, which are often based on the ISO/IEC 8859 standards, but alter them or extend them to suit their particular needs.

This wide variety of different encodings made multi-language communication difficult and confusing. As a solution, **Unicode** encoding has been widely adopted. The Unicode specification uses from one to four bytes per character, and allows over one million different symbols. The idea of Unicode is that it can encode any desired symbol without resorting to separate encodings, and it generally succeeds at this goal. Unicode contains characters that can encode most languages, including technical symbols and some ancient languages. There are different mechanisms for encoding characters as part of Unicode, but the most common are UTF-8 and UTF-16. In UTF-8, the size of a single character varies from 1 to 4 bytes, and the first few bits of the character indicate how long it is. It is a common encoding for Unix and Linux systems. In UTF-16, characters are either 2 or 4 bytes in size, and it forms the basis for internal character representation of most Windows versions and of the Mac OS.

While we don't want to get too technical with details of Unicode, we do want to mention a detail that can make a difference to a forensic examiner. In Unicode, given the huge number of symbols present, it is possible to find some symbols that look identical but are represented completely differently. One way this happens is if a character in one language is similar or identical in appearance to another. Another is if a symbol has an

Decimal	Hex Code	ASCII Character
32	0x20	Space
33	0x21	!
34	0x22	”
35	0x23	#
36	0x24	$
37	0x25	%
38	0x26	&
39	0x27	’
40	0x28	(
41	0x29	)
42	0x2A	*
43	0x2B	+
44	0x2C	,
45	0x2D	-
46	0x2E	.
47	0x2F	/
48	0x30	0
49	0x31	1
50	0x32	2
51	0x33	3
52	0x34	4
53	0x35	5
54	0x36	6
55	0x37	7
56	0x38	8
57	0x39	9
58	0x3A	:
59	0x3B	;
60	0x3C	<
61	0x3D	=
62	0x3E	>
63	0x3F	?
64	0x40	@
65	0x41	A
66	0x42	B
67	0x43	C
68	0x44	D
69	0x45	E
70	0x46	F
71	0x47	G
72	0x48	H
73	0x49	I
74	0x4A	J
75	0x4B	K
76	0x4C	L
77	0x4D	M
78	0x4E	N
79	0x4F	O
80	0x50	P
81	0x51	Q
82	0x52	R
83	0x53	S
84	0x54	T
85	0x55	U
86	0x56	V
87	0x57	W
88	0x58	X
89	0x59	Y
90	0x5A	Z
91	0x5B	[
92	0x5C	\
93	0x5D	]
94	0x5E	^
95	0x5F	_
96	0x60	‘
97	0x61	a
98	0x62	b
99	0x63	c
100	0x64	d
101	0x65	e
102	0x66	f
103	0x67	g
104	0x68	h
105	0x69	i
106	0x6A	j
107	0x6B	k
108	0x6C	l
109	0x6D	m
110	0x6E	n
111	0x6F	o
112	0x70	p
113	0x71	q
114	0x72	r
115	0x73	s
116	0x74	t
117	0x75	u
118	0x76	v
119	0x77	w
120	0x78	x
121	0x79	y
122	0x7A	z
123	0x7B	{
124	0x7C	|
125	0x7D	}
126	0x7E	~

Table 1.6: ASCII Codes for Printable Characters

accent mark. Some Unicode characters are accent characters that effect the character before it, stacking over or under it. Other characters include the accent. The same symbol can therefore be created in at least two different ways. This means that symbols that appear to be the same may not be, and documents that appear to be identical will not match up when compared bit by bit.

Encoding Dates

Forensic investigations focus on proving or disproving some hypothesis about what happened on a system. Very often, the occurrences of interest happened around a particular time or during a defined period. In these cases, the time and date information can be a vital part of the investigation.

Dates are stored in a simple binary integer encoding often called **timestamps**. Each stored date is a number in a counter that represents the amount of time that has passed since a particular base date and time. This base date and time is specific to an operating system, so it can vary from computer to computer. Most digital systems have a battery-powered clock that is used to set the counter when the system is started. Afterward, the counter is automatically incremented by hardware in the computer, frequently every millisecond ($\frac{1}{1{,}000}$ of a second) or microsecond ($\frac{1}{1{,}000{,}000}$ of a second). This gives a very fine granularity as to events on the system.

To demonstrate how a system like this works, let's take a look at a Windows XP time entry. The entry itself is 64 bits, and it represents the number of time intervals since January 1, 1601. (Which is very useful if you need to send a computer back in time to the 17th century.) Each time interval is 10^{-7} seconds, or $\frac{1}{10{,}000{,}000}$ of a second. A particular date and time is therefore a 64-bit binary integer value. Let's say that we have recovered a Windows XP timestamp with a hex value of 0x01C8F7ACC832C92C from a file of interest. We want to understand what that value means in a human readable date.

We first convert from hex to decimal, in this case using a calculator for speed and ease. We find that it is 128,624,910,845,266,220 intervals. Since each interval is one ten-millionth of a second, we can divide by that to find the number of seconds, which gives us 12,862,491,084, or over twelve billion seconds. Without showing all the math, that works out to 407.5964 years. Converting this value to dates (not forgetting that some years are leap years) from the base date of January 1, 1601, we will therefore end up with a date some time about halfway through 2008.

Now of course, computers process dates and convert them to human readable formats all the time, and use programs that do this much more easily than we can by hand, so let's not work out the exact date ourselves. Let's remember the point — that computer dates are binary integers representing some number of time intervals from a starting date — and then look at a program included on Windows that can decode the date for you. The program is called `w32tm` and is included on most Windows computers. To use it, choose `Start` → `Run`, then in the box that pops up, enter `cmd`. A window will appear where you can type commands.

In this window, first type:

```
C:> w32tm
```

Category	Read	Write
Owner	Y	Y
Administrator	Y	Y
Friend	Y	N
Others	N	N

Table 1.7: Simplified File Permissions

The program will provide a list of all the options. Examine the options for the `/ntte` setting. This setting will allow you to decode binary dates to a human-readable time. To find the exact date of the value above, type:

```
C:> w32tm /ntte 0x01C8F7ACC832C92C
148871 10:11:24.5266204 - 8/6/2008 6:11:24 AM (local time)
```

You can see that it shows the date in local time, so the result may be different depending on your time zone. The `148871` refers to the number of days since the base date; the `10:11:24.5266204`, hours, minutes, and seconds into the day; those values are followed by the local time.

There are often many caveats for using timestamps as part of investigations. For example, if the computer's timezone is set incorrectly or ignored by a program, we can be off by several hours. We'll return to this issue when we discuss standard investigative methods.

Encoding Flags

One of the types of data we often want to encode is the answer to a yes or no question. For example, we may want to keep track of who has access to a particular file on a system. For any user on a system the answer is either "yes", they can access the file, or "no", they can not. Just like a single bit, Yes or no is only two values. We can therefore store the answer to the question as a single bit, with a 1 meaning yes and a 0 meaning no.

Remember though that computers aren't good about tracking single bits, and instead work with bits in groups. While we can store the answer to a single boolean in a bit, we can store up to eight in a single byte, if we can examine the bits individually. Since we can a single byte can store eight boolean bits. When used in a way that each bit has a meaning, the bits are often called a **flag**. When used as a general data structure, the bytes holding the flags are referred to as a **bitset**. Let's look at a simplified example — real systems work this way, but are a bit more complex.

Suppose that we have four categories of users in relation to a particular file: owner of the file; system administrator; owner's friends; and everyone else. For each of these categories, we need to know who can read or write the file. Let's further suppose that we have a table of these permissions, as shown in Table 1.7.

For this particular file, the owner of the file can do whatever she wants with it, as can the administrator. Administrators commonly have access to all files for maintenance

Owner		Admin		Friend		Other	
Read	Write	Read	Write	Read	Write	Read	Write
Yes	Yes	Yes	Yes	Yes	No	No	No
1	1	1	1	1	0	0	0

Table 1.8: File permissions set as bits

reasons. The user is allowing her friend to read the file, but not write it, and all other users are prohibited from accessing the file.

Now we want to store this in binary form. The easier way to do so is to look at the table as a series of boolean values, as shown in Table 1.8. If we change each "Yes" to a one and each "No" to a zero, we can see that we have series of eight bits. Those can conveniently be stored in a single byte.

We can now convert the byte to a hex value of 0xF8. If we later need to find a particular file permission, we can extract it from the binary representation.

1.1.3 0x57 0x72 0x61 0x70 0x20 0x75 0x70

By now you should understand the most basic part of how a computer works: it converts all the data it uses into bits, and operates on those bits. Any information can be encoded into a bit string. Most often the encoding used represents some integer or decimal numbers, since those can be most easily and efficiently encoded and processed. All information in a computer is encoded this way, though we will often need some context to understand what the encoding is to recover the meaning of the data. Remember, it is all just bits!

1.2 How the Computer Processes Bits

A computer is just a tool — a tool that processes information. Like a hammer, computers are not inherently smart. They have no imagination. By themselves they can't think or learn. What they **can** do is to store and process bits, lots of them, and very, very quickly. This ability, combined with the right data and software to use for processing, allows them to perform very powerful tasks which can give the illusion of imagination, learning, and thinking.

In this section, we will give you an overview of the parts of a computer, and how it works together as a system. Our goal in doing this is twofold. First, as a forensic examiner you will need to know and recognize the basic pieces of hardware that make up a computer. Second, understanding the structure of the system will help you understand where potentially valuable data is located and can be found.

1.2.1 Computer Hardware

While the thought of mucking with the internals of a computer might seem intimidating, it shouldn't be. Computers are put together from common commodity parts by normal

people, most with just a little bit of training. Someone with no experience can build their own computer from parts pretty easily using guides from the Internet. Individual components are designed to snap together. Laptop computers can be a bit tougher to work with because the parts are packed into a tighter space, but with care and the proper tools they can still be disassembled and reassembled pretty easily. Consumer devices such as mobile phones or portable music devices, however, are often sealed to increase their durability and can be very difficult to open without damaging them.

However they are constructed, the vast majority of computers you will ever encounter will contain the same basic construction. They will contain a **central processing unit** or CPU, memory to store bits that are currently in use, and some long-term storage for other data. All of these components will be attached to a main circuit board that connects the major components using a **bus**, which provides electrical connectivity for transmission of the electrical signals used to represent bits. Let's look at the major components in turn.

The Central Processing Unit

The CPU is the heart of a computer. Everything else in the computer exists to keep the CPU fed with data and power so it can perform computation by manipulating bits, and the CPU can manipulate bits very, very quickly. The guts of the CPU are a **solid-state** device, which means that they have no moving parts, and instead work based on channeling electrical pulses though microscopic channels etched in silicon. While the internals of CPUs are incredibly complex, it is helpful to have a simple understanding of how they work.

Modern processors contain an internal clock that ticks at a rapid rate. During each tick of the clock, the CPU can complete a number of operations, including mathematical operations and logical operations. The set of possible operations depends on what choices the processor designer has made, as there is a trade-off between the complexity of implemented operations and how quickly the clock can tick.

On modern processors, the clock ticks incredibly quickly, up to four billion times per second. It is possible that such a processor could easily add together four billion different values and return the result in under one second. To put this in context, if you were adding up numbers yourself and were able to do one addition every second, it would take you more than 125 years to add those numbers up — without sleeping or eating. A processor can do more additions in one second than you could do in several lifetimes. It is really an incredible piece of technology.

The operations that the CPU runs are dictated by the **program** that is running. A program is a series of individual instructions that define which CPU operations are performed on which pieces of data. When you have many programs running on your computer at the same time, they are actually taking turns on the processor. A program will run for a while, until its turn is up or it requests data that isn't easily available. At that point the program is suspended and some other program gets a turn on the CPU. Because the CPU can switch programs very rapidly it provides the illusion of running many at the same time.

It used to be that each CPU was its own contained processor. Now, CPUs often incorporate many **cores** into a single chip. A core contains all the mechanisms of a single processor. Adding cores increases the number of processors available on the same CPU. This allows the CPU to truly run multiple programs at the same time, each on a different core. While no one program will run faster than before, many programs can run simultaneously, making the computer more efficient.

Each CPU, or each core in a CPU, has some space to store data it is working with. In general, though, this space is relatively small compared to the amount of memory and storage in the overall system. This is because creating storage space in the CPU is very expensive compared to storage elsewhere, since making room for storage reduces the space available for circuitry that does computation. The solution to this is to keep only the most important data locally, and then keep the rest elsewhere.

Endianness One technical aspect of CPUs that you will need to understand is the difference between **big endian** and **little endian** processors. The endianness of a processor is the order in which it stores the bytes. This concept is similar to the difference in languages that read left-to-right, like English which would be little endian, and right-to-left languages such as Arabic or Hebrew which would be big-endian.

Endian is a strange word indeed! Its origin is the book Gulliver's Travels, *which described a war between people that cracked open soft-boiled eggs from the little end and people that cracked open soft-boiled eggs from the big end.*

Number in english are Big-endian: the largest value (base 10) numeral is written on the left. A little-endian version of numbers in english would place the largest value numeral on the right. For example, the value four hundred twenty three is written 423 in Big-endian in english because the 4 numeral really a 400; the same value is 324 in a little endian of english.

Things are analogous for computers (which don't have a "left" or "right"). Big endian processors store the largest value byte in the largest address location in memory. Little endian processors store data so that the largest value byte is in the lowest address location in memory. Let's look at an example to really understand what this means.

In Section 1.1.2 above, we showed a time entry with a hex value of 0x01C8F7ACC832C92C. If we were to write this as individual bytes, we would see it was:

01 C8 F7 AC C8 32 C9 2C

This is big-endian format. The smallest value byte, 2C, is on the right in the smallest position. This is what we are naturally used to in base 10. If we were to transfer this value to a little-endian processor, we would have to reverse the order of the bytes to:

2C C9 32 C8 AC F7 C8 01

Compare those two hex values carefully — the **order of the bits in the bytes does not change, only the ordering of the bytes themselves**! It is not a reversal of the bit ordering. Why? Because 0xF represents only four bits, the nibble 1111_2. Endianness speaks to the ordering of the bytes in memory, not the ordering of the nibbles or bits. Here are some more examples of endian conversions:

Big endian	**Little endian**
01	01
01 C8	C8 01
01 C8 F7 AC	AC F7 C8 01
01 C8 F7 AC C8 32 C9 2C	2C C9 32 C8 AC F7 C8 01

You will most commonly encounter little-endian computer systems. Most common desktops and server systems use Intel or AMD processors based on the x86 architecture, which is little endian. There are still systems that are big endian, however: Internet packets are written in big-endian! Intel machines must carefully re-write the endianness of data before sending data to the network and after receiving it. Computer systems that used big-endian include older servers built by Sun Microsystems and older Macintosh computers that were based on the PowerPC processor. When in doubt, you will need to do research to ensure you are working with the proper endianness.

1.3 Overview of Cryptography

The nice thing about bits is that they are very easy to work with, given a computer. They can be changed very quickly, or copied between computers over the network or onto a wide variety of storage media. The bad thing about bits (sometimes) is that they can be easily changed or copied over the network or onto a wide variety of storage media. This is only bad when the bits represent something that you want to keep secure, or the person doing the alterations or copying is someone you don't want doing it. In that case you would prefer that your data be difficult or impossible for person to work with.

In this section, we are going to give you a brief overview of **cryptography**. We will show you how you can use cryptographic tools to ensure that data you have has not been altered in any way. We will also show you how to protect data from being revealed by encrypting it, and how encryption can be attacked.

1.3.1 Hashing

As forensic examiners, we will be working with data that can help prove or disprove a hypothesis about the actions of a computer user. Our work will have real impact on the lives of people who are involved in the outcome of the investigation, whether it is a criminal, civil, or an internal misuse case. Because the real-world stakes are high, forensic examiners have an obligation to ensure that the data they are working with is never altered. Any alterations would create the suspicion that the evidence had been tampered with. So how do we prove that nothing of the sort has happened?

One method we will use is to make sure that the data is kept in a physically secure location, so that it can't be accessed by unauthorized people. The second will be to keep track of all the individual bits we have copied, to ensure that not one has been changed. This sounds like quite the task, though, doesn't it? We may have billions or trillions of individual bits. How do we ensure that none of them are different than the original?

The answer lies in a cryptographic technique called **hashing**. The idea behind a hash function is that you can take any number of bits from a digital object, say a particular file or all the bits from a hard drive, and quickly and efficiently reduce them down to a small and fixed number of bits, typically less than 512. These reduced number of bits, which is often referred to as the **hash**, is like a fingerprint of that object. Good hash functions have some properties that ensure the hash is difficult to fake.

First, given a hash it is not possible to recover the original object. This means that you can share the hash with anyone, and they cannot tell what bits were given to create it. If you give them the original bits, though, it is easy to run those through the hash function to ensure that they are the same. Hashing is a one-way function.

Second, given a hash, it is exceptionally difficult to find any other bits that produce the same hash when run through the hash function. We will show what we mean by "exceptionally difficult" below with some examples, but for practical purposes it means impossible. In practice you cannot take a hash and find any other file that will hash in the same way.

Third, it is very difficult to even create two different objects that have the same hash, even if you don't have a specific hash in mind. It isn't as hard as finding a matching hash, but it is still very hard.

Fourth, changing any single bit in the input changes about one-half of the bits in the output. This means that the slightest alteration to the input changes the hash in a very noticeable way. We can therefore use hash functions to verify the integrity of large amounts of data, which is exactly what we need to ensure that our forensic images haven't been altered.

The way forensics examiners do this is to take hashes of any data that is collected while at the scene of the investigation. They then record this hash, typically on forms about the collection of the data and in their personal notes. The hashes become part of the case records. Later, when analyzing the data, the hash is recomputed and compared to the one recorded at the scene. If they match, and they should, then we can be assured that the copy represents an exact duplicate of the original.

In sum, here are the three most important properties of an ideal cryptographic hash function $H()$, which accepts as input a message m and outputs a value h.

1. Given h, it is hard to compute m such that $H(m) = h$.
2. Given specific m, it is hard to find another message m, such that $H(m) = H(m)$.
3. Given a large set of messages M, its hard to find any pair (m_i, m_j) that hash to the same value.

Hash Functions

Part of your toolbox as a forensic examiner will be one or more programs that can compute hashes. There are a variety of hash functions available now, and more will be created as time goes on. They all have the same purpose, which is to reduce a lot of bits to a few very representative bits. There are several that are currently in common use. They differ particularly in the size of the hash produced and in how secure they are. Table 1.9 shows some of the common hash functions and the size of their output. The output size, of course, relates to the number of possible hashes, and the table shows that number as a power of 2 and of 10.

Looking at the size of the output can help us understand how hashing works, in a practical sense. It may seem that given a hash you should be able to look for another

Hash Function	Output Bits	# Hashes	Approx. # of Hashes, power of 10
MD5	128	2^{128}	10^{38}
SHA-1	160	2^{160}	10^{48}
SHA-224	224	2^{224}	10^{67}
SHA-256	256	2^{256}	10^{77}
SHA-384	384	2^{384}	10^{115}
SHA-512	512	2^{512}	10^{154}

Table 1.9: Hash Functions and their Output Size

document that hashes to the same value. That is absolutely the case. You can make other documents and hash them and try to find one that matches. The problem is that it will take you close to forever to be successful.

Consider MD5, the hash with the smallest number of bits in the output. Even MD5 has a huge number of possibilities, though, with about 10^{38} total. For comparison, the size of the visible universe in meters is about 10^{26}. The number of hashes available in MD5 is about one trillion (1,000,000,000,000) times larger than the size of the whole universe in meters.

Every time you create a hash, you have a only a $\frac{1}{2^{128}}$ chance of getting a matching hash – which are truly, abysmally, awful odds. In fact, the chances of getting a match are less than he odds of winning the grand prize in the Powerball Lottery ($\frac{1}{195{,}249{,}054}$) *four times in a row.*

The difference between playing the lottery and hashing documents is that you don't have to pay a dollar to compute a hash. Since hashing is a fast operation, you can try many times cheaply on a computer. Let's say that you can create and hash 500 files a second, which is about how fast the system being used to write this handout on can do. That works out to:

$$\left\{\frac{500\ hashes}{1\ second}\right\}\left\{\frac{86,400\ seconds}{1\ day}\right\}\left\{\frac{365\ days}{1\ year}\right\} = \left\{\frac{15,768,000,000\ hashes}{1\ year}\right\}$$

On a single, relatively fast system we can do over 15 billion hashes in a year. To do enough hashes to cover the entire 128 bit space, then, it will take us:

$$\frac{\{2^{128}\ hashes\}}{\left\{\frac{15{,}768{,}000{,}000\ hashes}{1\ year}\right\}} = \{1\ year\}\left\{\frac{3.403 * 10^{38}}{15,768,000,000}\right\} = 2.16 * 10^{28}\ years$$

Or written out, that is $21,600,000,000,000,000,000,000,000,000$ years.

That is a long, long, long, long time. The good news is that we don't need to look at every single hash since once we find a match we can stop. On average, we'll stop half-way through, so we would expect to only need:

$$10,800,000,000,000,000,000,000,000,000\ years$$

Isn't that better? No?

So why did we go through this exercise? The point was to show you that while it is theoretically possible to find a hash that matches it is so unlikely as to be practically impossible, *assuming the hash function works correctly* (which is the next subject we'll tackle below). Given a 128-bit hash value, there are 2^{128} different possibilities, and that is such an incredibly huge number that you just can't possibly do any reasonable search of all possible hashes.

Notice as well that our example used MD5, which uses the smallest number of bits of the hash functions in our table. The other hash functions should be even harder to find matches for, since they use more bits and produce more hashes. Powers of 2 grow very large very fast. Once you are dealing with powers higher than 128, you are dealing with numbers that are so large that there is almost no way to comprehend them. For example, the entire universe is estimated to contain about 10^{80} atoms. The SHA-384 produces about 10^{115} hashes, or as many hashes as there are in all the atoms in 10^{35} universes. If you took every atom in the whole universe and write one possible SHA-384 value on each, you would run out of atoms long before you ran out of numbers. If it seems that we are going nuts with the analogies it is just to help you understand that these numbers are incredibly large, and there is effectively no chance of finding matching hashes — if the hash function is sound. Some, unfortunately, are not.

Collision Attacks Against MD5 All the analysis we just did was to show you that it is just about impossible to find a set of bits that hash to a specific value. This is how we expect our hash functions to work. Unfortunately, sometime the hash functions have flaws that are not obvious when they are proposed, reviewed, and adopted but which become obvious later.

MD5 is one of these hash algorithms that doesn't work as well as was thought. While it is apparently as difficult as we described above to find the match to a particular hash there is a different attack that is possible. In this attack, someone can create two different documents that have the same hash. This is different than what we have been talking about, since we were analyzing the odds of finding a hash that matches one we already have. The attack against MD5, known as a **collision attack**, requires the attacker to create two separate files that have the same hash — which is possible due to flaws in the way MD5 is constructed.

The result of this attack is that it is possible to create two bit sequences that have the same hash. Someone could give you something, say a preview of digital evidence that they had created and then provide you the hash of it. Later, they could switch to a different piece of evidence, and you would not be able to tell if all you had was the hash. Let's look at an example.

The media included with this handout includes two files, titled "order.ps" and "letter_of_rec.ps" (These files were created by Magnus Daum and Stefan Lucks of Ruhr-University Bochum). Files that end with a .ps are called PostScript files, and are designed to be printed. They are the forerunners of PDF files. These two files have completely different contents, if you examine them in a PostScript viewer. (You can download Ghostscript to help you view these files.) One is a letter of recommendation, the other an order to allow access to classified files. Yet, if we are to examine each of these files,

we will find that they have the same MD5 hash. Let's try it and see.

We will use the `md5sum` program to check these values, though any program that computes them should provide the same hash. There are a variety of command line programs that allow computations of MD5, and versions of `md5sum` are available for Windows, Mac OS X, and Linux.

If we look at these files, we see that they are each 2029 bytes long. When we compute the MD5 hash, they also seem the same:

```
% md5sum order.ps letter_of_rec.ps
a25f7f0b29ee0b3968c860738533a4b9 order.ps
a25f7f0b29ee0b3968c860738533a4b9 letter_of_rec.ps
```

If we hadn't looked inside, we would expect the files to be identical. But we know they are not. Let's see what another hash algorithm thinks. For this, we will use the OpenSSL version of SHA-1:

```
% openssl sha1 order.ps letter_of_rec.ps
SHA1(order.ps)= 3548db4d0af8fd2f1dbe02288575e8f9f539bfa6
SHA1(letter_of_rec.ps)= 07835fdd04c9afd283046bd30a362a6516b7e216
```

Now it is clear that if SHA-1 is working correctly then the files differ. And it is and they do. The problem is that MD5 is unreliable.

Because of this attack you should avoid using MD5. If you have to you should use a second hash function at the same time, as no one has been able to provide a collision for two different hash functions at the same time. If neither of those is possible, then never accept an MD5 hash for any file that you did not create yourself. But it shouldn't get to that, as other hash functions are common and free, and new ones that are more resistant to collision attacks are being worked on now.