4 Cryptography Goes Digital

Introduction

In this section you will learn about the next step in the evolution of cryptography, where it moves from ciphers that are performed on text, to ciphers that are performed on digital data. You will learn some of the basic building blocks of digital cryptography including how text is stored digitally as ASCII numbers, a math/electronics function called XOR (exclusive or) and how it's used to encrypt data using a one-time pad in something called a Vernam cipher, and how difficult it can be to generate truly random data to use as a OTP in a Vernam Cipher. You're not going to learn any new encryption schemes, since the OTP truly does provide perfect security if implemented correctly; you're just going to learn some of the basics of how the OTP is implemented digitally.

The specific things you should be able to do at the end of this section are:

1. Convert text characters to ASCII, specifically binary. And from binary/ASCII back to text.
2. Given a set of key bits, use the XOR function to encipher and decipher a set of bits.
3. List the differences between true random numbers and pseudo random numbers.
4. Use the modulo or remainder function to calculate whole number remainders.
5. List the requirements for a Cryptographically Secure Pseudo Random Number Generator (CSPRNG) and identify at least one CSPRNG.

Required Reading

Read this document first as it will provide you with the framework regarding all of the subjects covered in this section regarding digital cryptography.

In addition to reading this document you should read or view the following:

1. Read Singh chap 6 pgs 243 - 248

2. Go to the Khan Academy Journey Into Cryptography Website – https://www.khanacademy.org/computing/computer-science/cryptography

   Watch the videos and do the exercises under the Ciphers section:

   Ciphers vs. codes
   Shift cipher
   XOR bitwise operation
   XOR and the one-time pad
   Bitwise operators

   Watch the videos and do the exercises under the Ancient cryptography section:

   Frequency stability property short film
   How uniform are you?
   Pseudorandom number generators
   Random Walk Exploration

3. If you want more information or practice with Modulo or remainder arithmetic watch the following Khan Academy videos and do the exercises under the Modular arithmetic section:

   What is modular arithmetic?
   Modulo operator Practice
   Modulo Challenge

Optional Reading and Viewing
There are hundreds of web sites and web pages with information on cryptography. Here are a few of the more interesting and more useful sites that I've found over the years.

1. If you want more information regarding one-time pads and the XOR operation:

https://www.commonlounge.com/discussion/40916192c0514d99a72b9109bfb9b163

2. If you want more information regarding random numbers and randomness here is a site with dozens of links. You don't have to read any of this, but the links are there if you find this interesting and want to pursue it further:

   https://people.eecs.berkeley.edu/~daw/rnd/ - dozens of links to random sites (excuse the pun)

3. Sites with information about generating random numbers and random bits.
   https://csrc.nist.gov/projects/random-bit-generation
   https://en.wikibooks.org/wiki/Cryptography/Random_number_generation

4. If you want to watch Christof Paar, a really smart guy, provide a lecture on this subject. He goes into depth with the math while writing on a chalkboard. If you have a good background in math and you want to know the details behind some of the algorithms you should find this interesting.

   https://www.youtube.com/watch?v=AELVJL0axRs

Section Content

Introduction

At this point you've learned about text ciphers, going from very simple rotation ciphers to more complex poly alphabet ciphers like the Trithemius and Vigenère ciphers. You've also learned that under certain circumstances the Vigenère cipher can provide perfect security. This is done by using a One Time Pad, where the key is as long as the plain text, and the key is only used once.

It's important to note that if One Time Pads are used to encrypt a text message, then it's game over for anyone trying to crack the encrypted messages. It doesn't matter if the attackers use a computer, or even a quantum computer, there would be no way to discern the original message. You need to also note that this only applies to *text* messages. Of course, this made sense when the encryption and decryption were being done by humans who had to do the work manually, and in a time before storing other types of data digitally had been invented. But the thing to recognize is that if the only concern is encrypting *text* messages the class could end here, there'd be nothing else to learn.

However, as you can probably guess, there's still more to learn about cryptology. The main reason for this is those darn computers. Just using computers for cryptography obviously brought huge improvements to the speed at which messages could be encrypted or decrypted. But another impact of using computers is the fact that computers store all data, including text, in a digital or in binary format. With modern cryptography we want to work with all kinds of data. This still includes text, but we also want to encrypt and decrypt image files, audio files, databases, etc., all different types of digital data. Luckily, the main concept of the Vigenère cipher and One Time Pads still applies. That is, if we use an encryption scheme that constantly modifies the encryption alphabet it will be unbreakable. We just have to adapt it to work on data stored in a binary format. This might sound simple, and technically speaking it's pretty straightforward. But there are a few details that need some explanation.

To help you make the jump from encrypting text to encrypting binary data we will break this chapter down into a few main parts:

1. **Storing Data Digitally and ASCII**. The first is background on how different types of data are represented digitally. Hopefully you already have at least a surface understanding of how this is done for the different data types. But just in case you need a refresher we will go into detail looking at how text is stored digitally. We won't go into detail for other types of data like images or sound files, and you don't really need to know these details for this class. But I will provide you links to other resources just in case you're interested.

2. **Vernam Cipher and XOR**. With digital cryptography, once we have our data in digital format, we will do something like what we did with the Vigenère Cipher. That is, with the Vigenère cipher we took the plain text character, then used the character from the key to determine the cipher text character. Since we're now using digital data, we can't do exactly the same thing, but we're going to use something called the Vernam cipher that's similar in a general sense. The second main subject you will learn about in this section is the Vernam cipher. But this will require a side trip where you learn about Boolean logic, the XOR function, and how XOR is used in the encryption and decryption processes.

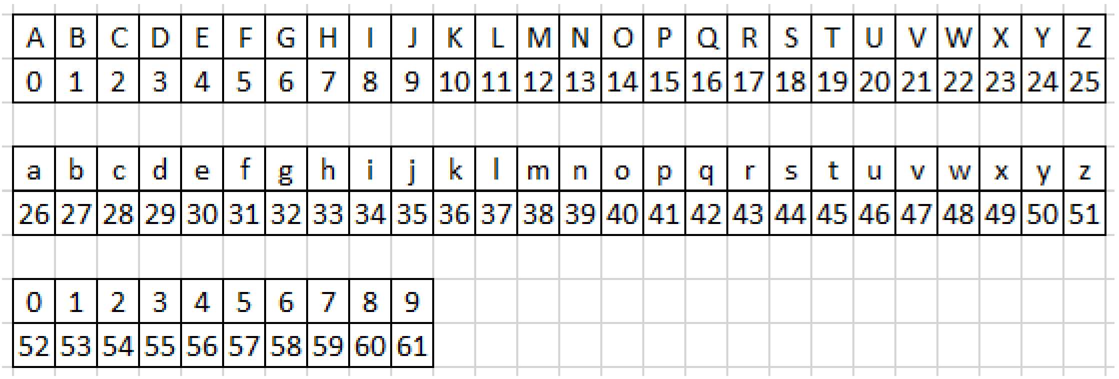
3. **Digital One Time Pads and Random Numbers**. The last major item you will learn about is how we try to simulate one-time pads digitally. However, in this case the OTPs won't be strings of characters since we're dealing with binary data. In this case the one-time pads will be sequences of ones and zeros. You'll learn about how it's difficult to generate a truly random set of bits and take a bit of a deep dive into random numbers, and how they are generated.

This section will end with a summary, where all these parts are brought together.

How text is converted to numbers using the ASCII Table

The first building block of digital cryptography to learn about is the method for representing text characters as numbers. It may seem like the obvious answer to this problem is to simply build a table to assign each character a number, and then convert that number to binary. For example, using decimal numbers, A could be assigned the number 0, B could be assigned the number 1, C could be assigned the number 2, etc.

And unsurprisingly this is exactly how text characters are converted to digital numbers. The only trick is deciding what table to use, and then getting everyone involved to agree to use the same table. For example, since I'm a geek and like to start counting with 0 my system might use the following table:

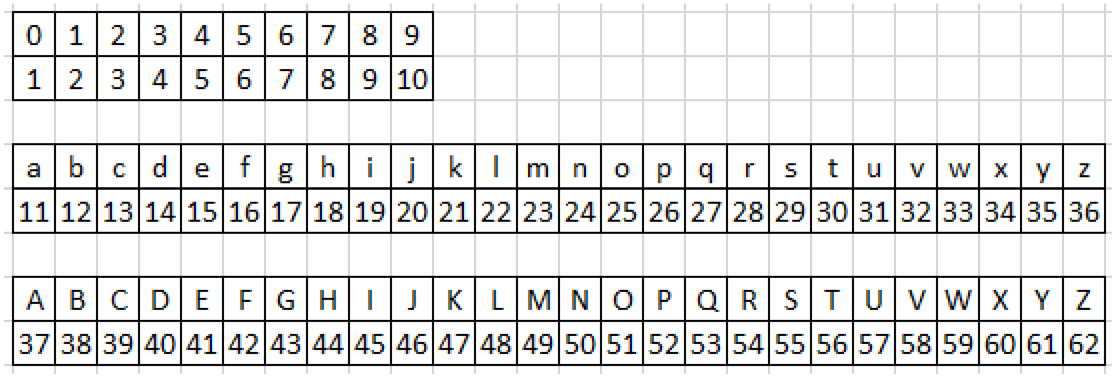


A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

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
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51

0	1	2	3	4	5	6	7	8	9
52	53	54	55	56	57	58	59	60	61

But other people may want to start with the numbers first, then lower case letters; and maybe they want to start counting with the number one instead of zero. They might want to use a table that looks like the following:



0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

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
11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36

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
37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62

This was a problem faced by the first software developers as there were thousands of potential numbering systems. But if they wanted to exchange data with others, and have the recipient read the data it would only work if the sender and the recipient used the same numbering system.

During the early years of computing many different systems were designed and used to convert characters to numbers for use in computing. But in the 1970's it came down to 2 different options, ASCII (American Standard Code for Information Interchange)[1][2], and EBCDIC (Extended Binary Coded Decimal Interchange Code)[3] which was IBM's standard. ASCII became the de facto standard when it was adopted by Apple and Microsoft and other industry players, and then it became an actual legal standard when it was adopted by the International Standards Organization (ISO). ISO didn't actually choose ASCII themselves, but when they were faced with developing a system for enumerating every character and symbol used by human kind, they realized it was an immense task and wisely decided to turn to an organization named UNICODE whose only reason for existence was to map every possible character to a number. And UNICODE decided that for the plain text characters and symbols on a standard keyboard they wouldn't reinvent the wheel, and wisely chose to use the ASCII encodings.

Details on ASCII

The table that (almost) everyone and every system uses to convert text to binary or digital data is the ASCII table. If you say the full name, American Standard Code for Information Interchange, no one will know what you're talking about, so just say ass-key. Let's take a quick look at the table so you get a general sense of how it works. You'll see the table itself is pretty straightforward. But there are a few oddities, like the first 32 characters, which could probably use some further explanation.

The first thing to note is that the actual text character mappings don't start until number 32, which is the space character. This is followed by the "special characters" like bang for number 33, double quotes for 34, etc. The upper-case alpha characters start with number 65 for **A**, and run through number 90 for **Z**. The lower-case alpha characters start at 97 for **a** and run through 122 for **z**.

[1] https://www.youtube.com/watch?v=5aJKKgSEUnY - Understanding ASCII and Unicode (GCSE)

[2] http://www.asciitable.com/ - A good copy of the ASCII table

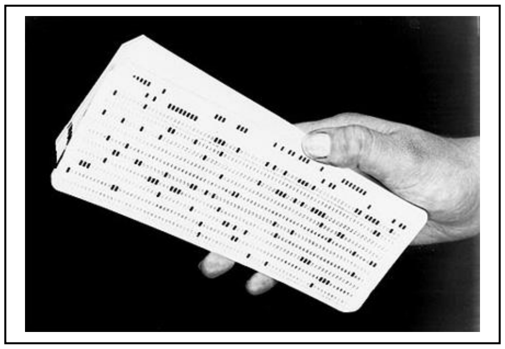
[3] http://www.lookuptables.com/ebcdic_scancodes.php - If you want to see what EBCDIC looks like.

Dec	Hex	Binary	
0	0	00000000	Null
1	1	00000001	Start of Heading
2	2	00000010	Start of Text
3	3	00000011	End of Text
4	4	00000100	End of Transmission
5	5	00000101	Enquiry
6	6	00000110	Acknowledgment
7	7	00000111	Bell
8	8	00001000	Back Space
9	9	00001001	Horizontal Tab
10	A	00001010	Line Feed
11	B	00001011	Vertical Tab
12	C	00001100	Form Feed
13	D	00001101	Carriage Return
14	E	00001110	Shift Out / X-On
15	F	00001111	Shift In / X-Off
16	10	00010000	Data Line Escape
17	11	00010001	Device Control 1
18	12	00010010	Device Control 2
19	13	00010011	Device Control 3
20	14	00010100	Device Control 4
21	15	00010101	Negative Acknowledgement
22	16	00010110	Synchronous Idle
23	17	00010111	End of Transmit Block
24	18	00011000	Cancel
25	19	00011001	End of Medium
26	1A	00011010	Substitute
27	1B	00011011	Escape
28	1C	00011100	File Separator
29	1D	00011101	Group Separator
30	1E	00011110	Record Separator
31	1F	00011111	Unit Separator

Dec	Hex	Binary	
32	20	00100000	space
33	21	00100001	!
34	22	00100010	"
35	23	00100011	#
36	24	00100100	$
37	25	00100101	%
38	26	00100110	&
39	27	00100111	'
40	28	00101000	(
41	29	00101001	)
42	2A	00101010	*
43	2B	00101011	+
44	2C	00101100	,
45	2D	00101101	-
46	2E	00101110	.
47	2F	00101111	/
48	30	00110000	0
49	31	00110001	1
50	32	00110010	2
51	33	00110011	3
52	34	00110100	4
53	35	00110101	5
54	36	00110110	6
55	37	00110111	7
56	38	00111000	8
57	39	00111001	9
58	3A	00111010	:
59	3B	00111011	;
60	3C	00111100	<
61	3D	00111101	=
62	3E	00111110	>
63	3F	00111111	?

Dec	Hex	Binary	
64	40	01000000	@
65	41	01000001	A
66	42	01000010	B
67	43	01000011	C
68	44	01000100	D
69	45	01000101	E
70	46	01000110	F
71	47	01000111	G
72	48	01001000	H
73	49	01001001	I
74	4A	01001010	J
75	4B	01001011	K
76	4C	01001100	L
77	4D	01001101	M
78	4E	01001110	N
79	4F	01001111	O
80	50	01010000	P
81	51	01010001	Q
82	52	01010010	R
83	53	01010011	S
84	54	01010100	T
85	55	01010101	U
86	56	01010110	V
87	57	01010111	W
88	58	01011000	X
89	59	01011001	Y
90	5A	01011010	Z
91	5B	01011011	[
92	5C	01011100	\
93	5D	01011101	]
94	5E	01011110	^
95	5F	01011111	_

Dec	Hex	Binary	
96	60	01100000	`
97	61	01100001	a
98	62	01100010	b
99	63	01100011	c
100	64	01100100	d
101	65	01100101	e
102	66	01100110	f
103	67	01100111	g
104	68	01101000	h
105	69	01101001	i
106	6A	01101010	j
107	6B	01101011	k
108	6C	01101100	l
109	6D	01101101	m
110	6E	01101110	n
111	6F	01101111	o
112	70	01110000	p
113	71	01110001	q
114	72	01110010	r
115	73	01110011	s
116	74	01110100	t
117	75	01110101	u
118	76	01110110	v
119	77	01110111	w
120	78	01111000	x
121	79	01111001	y
122	7A	01111010	z
123	7B	01111011	{
124	7C	01111100	|
125	7D	01111101	}
126	7E	01111110	~
127	7F	01111111	Del

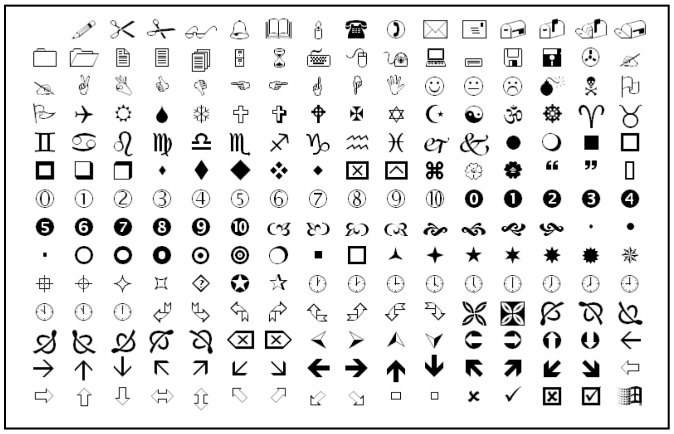


And now for the first 32 characters. What the heck is the **Null** character or the **Start of Heading** character? The first 32 characters are an anachronism, leftovers from the 1960s and 1970s when ASCII was being developed. This was back in the days of the dinosaurs, when computers were big and expensive, and it wasn't possible to own a computer or in many cases even communicate directly with the computer you were using. Computer users created decks of punch cards to run their programs and store their data. These decks of cards would be fed into a card reader, which would send the program instructions and data to the computer for processing. Since all the cards were fed in at one time there needed to be some way to tell the computer which set of cards contained the instructions and which cards contained the data files. Some of the first 32 ASCII characters, like 28 **File Separator** are used to accomplish this. And in those days output wasn't sent to a video screen, it was sent to a printer. When the computer would print out the results, they needed ways to encode some commands like sending a carriage return to a printer, or having the printer ring a bell to let the operator know a print job was complete. The ASCII characters between 0 and 32 are called non-printable characters, because they don't print, instead they were designed for controlling input and output.

7 Bit and 8 Bit ASCII

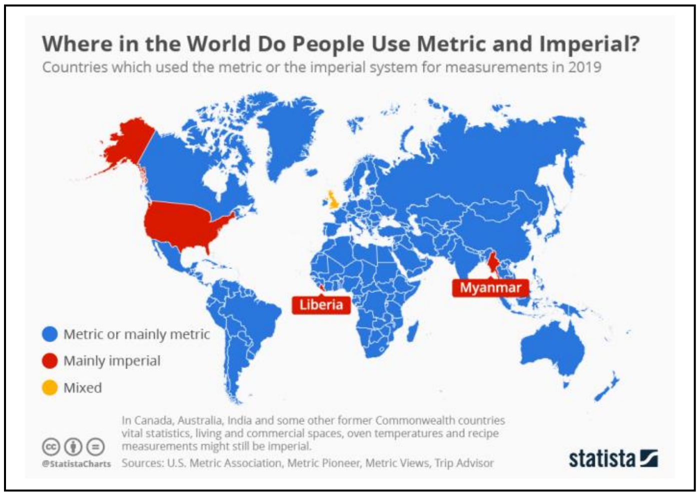
One thing to note about ASCII is you might hear about versions called 7-bit ASCII and 8-bit ASCII. The different versions came about in the early days, when ASCII was first being defined. The original ASCII mapping only used 7 bits. This allows for 128 decimal numbers which is enough to map all the text characters on a normal English computer keyboard to binary. It actually only takes 94 numbers to map all of the characters on a standard keyboard as shown in the following table, but remember ASCII also contains mappings for the non-printing characters.

Upper Case Alpha (ABC …)	26
Lower Case Alpha (abc …)	26
Numerals (0-9)	10
Special Characters (~`!@#$ …)	32
Total	**94**



The original ASCII only mapped characters to these 7 bits, which is how it got the name 7-bit ASCII. But since computers typically use 8 bits or 1 byte to store data, so there's also an 8-bit ASCII which may also be called extended ASCII. With 8 bits it's possible to map 256 different characters, or 128 more than with 7-bit ASCII. In the early days these ASCII mappings for character numbers 128-256 were undefined and left up open to any software developer to define as they saw fit. The software developer could map the numbers to symbol characters like Wing-Dings, or characters from non-English character sets, or drawing symbols

… anything they wanted to use. This meant that if you created a document in one application and used characters in the 128-256 range, then displayed that document in another application the results could be unpredictable. However, later ASCII locked down the definitions for character numbers 128-256.


Where in the World Do People Use Metric and Imperial?
Countries which used the metric or the imperial system for measurements in 2019
Liberia
Myanmar
Metric or mainly metric
Mainly imperial
Mixed
In Canada, Australia, India and some other former Commonwealth countries vital statistics, living and commercial spaces, oven temperatures and recipe measurements might still be imperial.
@StatistaCharts
Sources: U.S. Metric Association, Metric Pioneer, Metric Views, Trip Advisor
statista

You might think the ASCII numbering scheme is a little goofy, and I'm sure you could find plenty of people to agree with you. But it's the numbering scheme that almost every OS and piece of software uses so it would take a huge effort to change it. It's like saying that driving on the right side of the road is goofy. Or, using the Imperial system of weights and measures is goofy and we should all use metric. The switch to the metric system actually makes sense and in 1975 the US passed a federal law, The Metric Conversion Act, which declared metric as the preferred system. And President Ford signed Executive Order 12770, which directed all US Federal Departments and Agencies to "take all appropriate measures within their authority" to use the metric system. Even though this switch makes perfect sense and was required by law it never happened. We still buy gas by the gallon and our road signs are all measured in miles. While I personally think it's ridiculous and a little pathetic that the US doesn't use the metric system isn't the reason I brought this up. The reason I'm pointing this out is that even though ASCII may seem a little goofy and could be improved, changing ASCII at this point in time would be like changing 2x4's to 40mmx80mm or buying milk by the liter instead of by the gallon. ASCII is now so entrenched and embedded in so many systems that it would be almost impossible to change.

Now let's look at the part of the ASCII table we really came here to see, the part that's pertinent to cryptography, which is the binary representation of text characters. Here are a few examples of using the ASCII table to convert text to binary, and then binary back to text.

Say we want to convert the text **I am the Walrus** to binary. The ASCII table shows the binary value for the character **I** is 01001001_2. The binary value for the space character is 00100000_2, which is followed by **a** which has the binary value 01100001_2. Repeating this process for the entire string results in:

I - 01001001
<space> 00100000
a - 01100001
m - 01101101
<space> 00100000
t - 01110100
h - 01101000
e - 01100101
<space> 00100000
w - 01010111
a - 01100001
l - 01101100
r - 01110010
u - 01110101
s - 01110011

To be honest I only converted the first 3 characters manually. To speed things up after that I used an online tool to do the rest of the conversion for me.

Once again, the important thing for you to understand is that in modern cryptography the algorithms work on binary data, and the ASCII table is the standard way for converting text characters to binary.

If you look at most ASCII tables on the Internet, you won’t see the binary representation. You’ll see the decimal and maybe hex numbers associated with each character, but not the binary. This is because the binary numbers are hard for most humans to understand, and they’re relatively long so they take a lot of extra space to display in the table. But computers handle binary with ease, and as you’ll see digital cryptography works on binary numbers so it’s important that you have at least a surface understanding of how computers store each text character as a binary number.

While you need to know that the ASCII values are being used to represent text in binary. You don’t have to memorize the ASCII table or any of the character numbers, and you don’t really have to be able do the conversion yourself. You can either find a table, like the one I’ve given you that shows you the ASCII values in binary, or if you only have the decimal values you can convert them to binary yourself. You’ll have to do this for a few of the assignments and test questions in this class, so it’s helpful if you know how to do the conversion yourself. And if you are serious about a career in Cyber Security or Computer Science you should be able to work with at least 8-bit binary numbers. Hopefully you remember enough binary to understand 8-bit binary numbers and are able to convert between decimal and binary for 8-bit binary numbers or decimal numbers between 0-127. If you need to review binary there are plenty of online references including:

https://www.electronics-tutorials.ws/binary/bin_2.html
https://www.youtube.com/watch?v=JUd_2RD5wNo

If these sites no longer exist there should be many alternatives that can be found by searching for something like “binary math tutorial”.

Converting from hex to binary

Most ASCII tables won’t show the binary number for each character, but they will show the hexadecimal number. But to use the numbers in cryptography we need the binary version. Luckily there’s a simple trick for converting between hex and binary. Once you learn the trick, performing the conversion between hex and binary becomes simple and quick, much easier than it is to convert between decimal and binary. The quick explanation of the trick is to treat each hex digit has a four digit binary number. Or to go the other way treat each group of four bits as a single hex number.

If the quick explanation didn’t make sense, here’s a longer explanation and a quick refresher on hexadecimal, plus details and examples of converting between hex and binary.

Remember that hexadecimal is base 16 and has the numerals 0 through 9 just like decimal, and then uses the letters A through F to represent 10, 11, 12, 13, 14, and 15. If you look at a four digit binary number it can also represent decimal numbers 0 through 15. So, to convert from binary to hex, just figure out the decimal value, then convert this to hex. For example, if you have the binary number 1001_2 you can probably convert this pretty quickly in your head to 8 + 1 or 9. So 1001_2 in binary is 9_{16} in hex. Or 1100_2 in binary is equal to 8 + 4 in decimal, and 12 in decimal is C_{16} in hex.

The trick for converting larger numbers is to just break them into 4 bit chunks, and then convert each 4 bits into a separate hex character. For example, converting $1001\ 0001\ 1011\ 0110_2$ to hex might seem daunting. I know I would have a very difficult time converting the entire number to decimal, and then to hex. But if it's broken down into 4 bit chunks it's no problem at all because you never have to count higher.

Running through the example we get:

1st four bits: $1001_2 = 9_{10} = 9_{16}$
2nd four bits: $0001_2 = 1_{10} = 1_{16}$
3rd four bits: $1011_2 = 11_{10} = B_{16}$
4th four bits: $0110_2 = 6_{10} = 6_{16}$

Or:

$1001\ 0001\ 1011\ 0110_2 = 91B6_{16}$

Here's an example of going the other way, and converting the hex number D4A7 to binary:

$D_{16} = 13_{10} = 1101_2$
$4_{16} = 4_{10} = 0100_2$
$A_{16} = 10_{10} = 1010_2$
$7_{16} = 7_{10} = 0111_2$

Or:

$D4A7_{16} = 1101\ 0100\ 1010\ 0111_2$

Hopefully you understand this conversion trick as it can really help you if you ever need to do the conversions manually. But if you don't understand the trick, don't worry; you can always use an online tool to convert numbers between the various number systems. You can also use a calculator like the Windows calculator. Just make sure that it's in Programmer mode, which is the mode that allows you to enter numbers in one base and will automatically convert them to a different base.

Converting other types of data to digital

You've just learned how text is converted to binary for use in digital systems, but as I'm sure you know computers use binary to store all their data. So, any encryption scheme that will work with text should work with any other kind of data such as images, music, etc. The computer doesn't care what the data represents, it just treats everything as 0's and 1's. There isn't the time or space here to provide you with the details on how every type of data is stored digitally, but if you’re interested here are some references to get you started learning about this subject. (If these links don’t work, do your own search.)

Images:

https://www.bbc.co.uk/bitesize/topics/zf2f9j6/articles/z2tgr82
https://www.imaging.org/site/IST/Resources/Imaging_Tutorials/How_a_Picture_can_be_Represented_as_a_Collection_of_Numbers/IST/Resources/Tutorials/Digital_Image.aspx?hkey=53cc8b2f-3ca0-4ace-93d2-459645ed8554
https://www.fileformat.info/format/gif/egff.htm
https://www.imaging.org/site/IST/Resources/Imaging_Tutorials/What_s_Inside_a_JPEG_File/IST/Resources/Tutorials/Inside_JPEG.aspx?hkey=f9946f90-9f14-452d-897c-ac1612116e2d

Audio:

https://manual.audacityteam.org/man/digital_audio.html
http://artsites.ucsc.edu/EMS/Music/tech_background/TE-16/teces_16.html

http://sites.music.columbia.edu/cmc/MusicAndComputers/chapter2/02_01.php

Video:

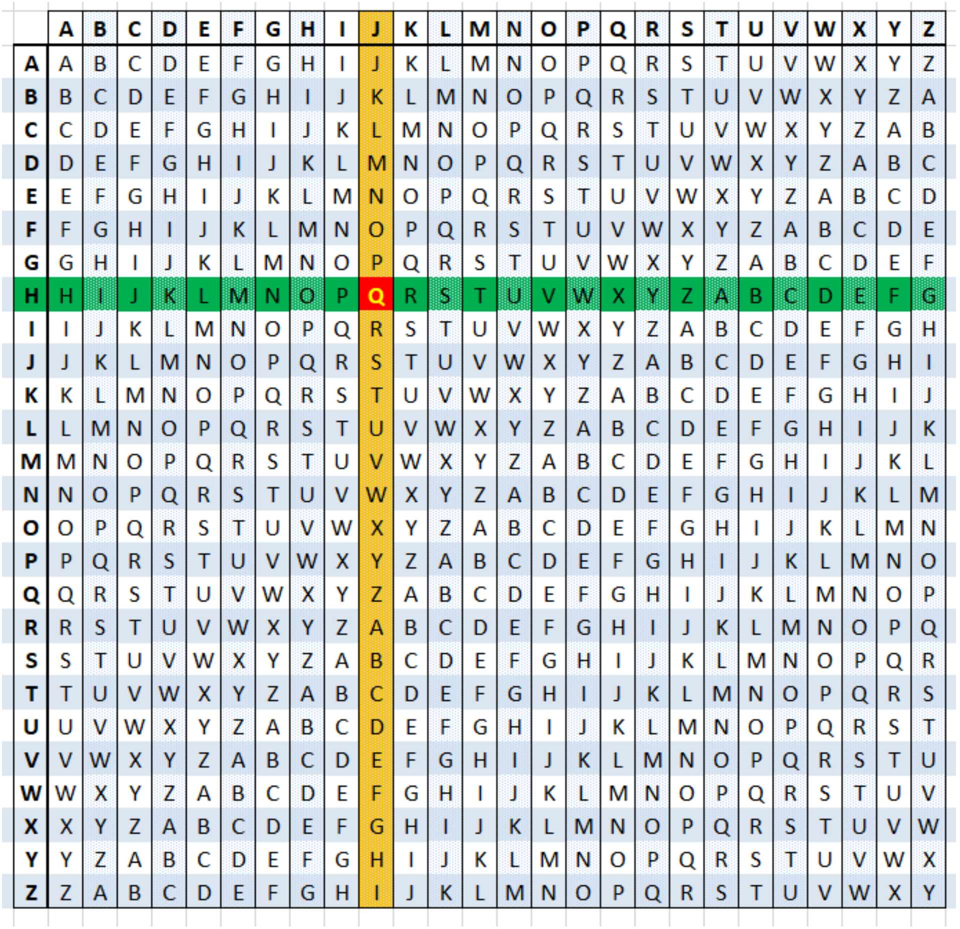
https://elearningindustry.com/how-digital-video-works-digital-video-101

XOR function

Now that you understand how text characters can be converted to binary numbers by using the ASCII table, the next critical piece in understanding digital cryptography is to learn about something called the XOR function.

Before we go into the details of the XOR function, let's do a quick review of the Vigenère cipher, because that will help explain the part the XOR or function plays in digital cryptography.

Remember that the first step in using the Vigenère cipher is creating the 26 by 26 table with the different alphabets. The next step is to define the key word or key phrase and then use this key phrase to determine which row of the table to use to encrypt each plain text character. Or if we look at this another way, we take the plain text character and combine it with the character from the keyword to determine the cipher text character. For example, the following figure shows how the plain text characters **J** is combined with the keyword character **H**, which results in the cipher text character **Q**.



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

To decrypt the cipher text encrypted with the Vigenère cipher, you need to know the encrypted text character and the character from the key, and this will lead you back to the plain text character. If you only know the encrypted text character and that it's been encrypted with the Vigenère cipher, you can't get back to the plain text character unless you know the character from the key.

With digital cryptography we still want to do the same process, or at least a similar process, with our binary data. But since we've replaced the text characters with zeros and ones, we're not going to be able to use the same table as we did with the Vigenère cipher. We can use a similar process, but we can't use the exact same table.

Of course, we could still perform the encryption and decryption of digital text by converting the binary numbers back to their equivalent text characters and then using the same table, but this process causes two problems. The first problem is that this would be super limiting because it would only work for text characters, and only in either upper-case or lower-case but not both.

You could expand the Vigenère table to include more characters, but even adding numbers and special characters would make the table very large and unwieldy. The size of the table wouldn't be a problem with modern computers, but on a computer we can represent any kind of data digitally, like the bits from a sound file or a picture file, and the Vigenère table can only handle text characters.

The second problem is that using a giant table would be relatively slow if we had to encrypt and decrypt manually. On *modern* computers the encryption and decryption could be done very quickly, even for long messages. But when computers were first being developed, and up through the 1980's and 1990s programmers did everything they could to reduce the number of program instructions required to accomplish a task as each instruction required extra time to execute.

Even using an ancient computer from the 80's or 90's to do the encryption and decryption with a Vigenère table would be much faster on a computer than it would be doing it by hand, but it turns out that there is a simple programming method called XOR, or e**X**clusive **OR**, that can be used to obtain the same general result as the Vigenère table but it works with binary data so it isn't limited to text. Plus, the XOR function has the benefit of executing very, very quickly as modern CPUs have an XOR instruction as part of their base instruction set[4].

The XOR function is part of a family of Boolean logic commands that are often used in programming. If you've done any programming, you're probably familiar with Boolean logic for deciding whether something is TRUE or FALSE, and combining truth decisions using the Boolean AND and OR operators. Let's do a quick review of Boolean AND and OR before looking at XOR.

Remember that in C based languages the AND operator is represented by && while the OR operator is represented by ||. Also remember that AND looks at two items, and if both are TRUE, then the result is TRUE. If either of the items is false, then ANDing them together will return a FALSE. For example, if you have a programming statement like:

```
if (weight > 120) && (total < 2300)
```

[4] https://en.wikipedia.org/wiki/X86_instruction_listings

The end result will only be TRUE if both weight > 120 **and** total < 2300 are TRUE.

The OR function looks at two items and will return TRUE if either of the items is TRUE. The only time OR returns a FALSE is if both items are FALSE. For example, if you have a programming statement like:

```
if (weight > 120) || (total < 2300)
```

The end result will be TRUE if either weight > 120 or total < 2300 is TRUE. If both statements are TRUE ORing them together will result in a TRUE. The only way to get a result of FALSE is if both items are FALSE.

Now let's look at the XOR function. The XOR function's full name is Exclusive OR and it's similar in behavior to the OR function, with one important difference. XOR only returns a TRUE if only one of the items is TRUE. If both items are TRUE XOR returns a FALSE. This is unlike OR, which return TRUE if both items are TRUE.

Here's an analogy that may help you understand how XOR is different than OR. Say you're taking your nephew out for something to eat. You finish the main meal and ask him if he wants dessert. The dessert choices are apple pie OR ice cream OR cheesecake. If he chooses any of these the answer to the dessert question will be TRUE, he wants some dessert. But in reality, his mom, your sister, will be very upset if you let him eat three desserts. You need your nephew to choose just one dessert so to be mathematically accurate the question should be does he want apple pie XOR ice cream XOR cheesecake. This means that your nephew will get dessert if he chooses one item, but if he chooses two or three items he won't get any.

To understand how this helps in cryptology it helps to look at the truth table for XOR. A truth table is another way to look at these Boolean functions and what they return when comparing two items. With truth tables 1 represents TRUE and 0 represents FALSE. The possible values for item 1 are written across the top row while the possible values for item 2 are written down the first column. The table cells hold the result of combining the row item and the column item using the specified Boolean operator, AND, OR, or XOR. The truth tables for binary items are super simple because there are only two possible values for any item, 0 or 1.

For example, the truth table for the AND function looks like this:

AND	Item 1 value = 0	Item 1 value = 1
Item 2 value = 0	0	0
Item 2 value = 1	0	1

To use the truth table, combine the column item with the row item, then look in the table cell for the result. For example, here's how to use the table to see the result of 1 AND 0, which is 0 or FALSE.

AND	Item 1 value = 0	Item 1 value = 1
Item 2 value = 0	0	0
Item 2 value = 1	0	1

The truth tables are usually written a little more concisely, without the long table headings. For example, the truth table for AND usually looks like this:

AND	0	1
0	0	0
1	0	1

The truth table for the OR function looks like this:

OR	0	1
0	0	1
1	1	1

The truth table for the XOR function looks very similar to the truth table for OR, the only difference is when both item 1 and item 2 are TRUE the result is FALSE:

XOR	0	1
0	0	1
1	1	0

The reason it's called the exclusive or, is because it's only TRUE when only one item is TRUE, and it excludes the case when both are TRUE.

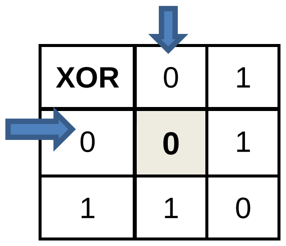
I know the truth tables might seem super simple, but at the same time might seem like a bunch of confusing gibberish. But don't worry about absorbing all the details of all the tables, just make sure that you understand how to read the truth tables as they're a good tool for visualizing what happens with digital encryption.

So … how does this apply or relate to cryptology and Vigenère cipher? Remember that in general terms the Vigenère cipher combines two items to calculate a third item. That is, you combine a plain text character with a character from the key to get the cipher text character. If you have the cipher text character, the only way to get back to the first item is if you also know the second item. Knowledge of the cipher text character doesn't do you any good on its own, you can only decrypt it if you know the character from the key.

With digital cryptography we want a way to accomplish the same thing as the Vigenère cipher, but we need a way that deals with bits of binary data. We need a way to combine our starting bit (plain text) with a second bit (from the key), to get a third bit (the cipher text). The other requirement is that even if someone knows the third bit (cipher text) they can't deduce the starting bit (plain text) unless they know the second bit (key bit). This is where the XOR function comes in to provide a perfect solution by simulating the Vigenère cipher process, but with binary data. This digital version of the Vigenère cipher is called the Vernam cipher after it's inventor, Gilbert Vernam who patented his process in 1917[5].

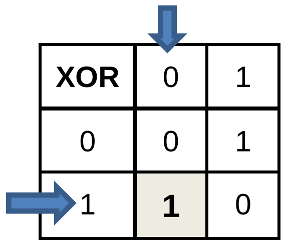
[5] https://www.hypr.com/vernam-cipher/

Visualizing this is very simple with XOR table and binary data since there are only 2 possible values for the plain text bit, and 2 possible values for the key bit. That is, it's simple because there aren't 26 possible values and a 26x26 table like there are with text characters and the Vigenère table. There are only 2 possible values, 0 and 1, and a 2x2 table. Once the table is set up it's used just like the Vigenère cipher table. For example, let's say the plain text bit is a 0 and the key bit is also a 0. To look up the cipher text bit value in the table, move to the column specified by the plain text bit, and the row specified by the key bit. The table cell at the intersection of the column and the row has the value of the cipher text bit. In the case of the following example the cipher text bit would also be a 0.



XOR	0	1
0	**0**	1
1	1	0

Or if the plain text bit is a 0 and the key bit is 1 the cipher text bit would be a 1 as shown in the following figure.



XOR	0	1
0	0	1
1	**1**	0

Once again, the XOR table is a lot like the Vigenère cipher table, just much smaller and much simpler.

The XOR table also functions like the Vigenère cipher table for decryption. That is, even if you know the value of the cipher text bit, and you know the structure of the table, the only way to determine the value of the plain text bit is if you know the value of the key bit. For example, if you know that the cipher text bit is a 0, the value of the plain text bit could be 0 or 1, either one is equally possible.

XOR	? 0	? 1
0	**0**	1
1	1	**0**

The only way to be certain of the value of the plain text bit is if you know the value of the key bit. If the value of the key bit is 0, then the value of the plain text bit is 0.

XOR	**0**	1
0	0	1
1	1	0

But if the value of the key bit is 1, then the value of the plain text bit is 1.

XOR	0	**1**
0	0	1
1	1	0

The Vernam cipher and the XOR table are very simple, but I sometimes think that this simplicity can be a source of confusion. After all, how can something so simple possibly replace the Vigenère cipher and the Vigenère cipher table which were much larger and more complicated? And how can the Vernam cipher possibly provide the same level of security as the Vigenère cipher and the One Time Pad (OTP)?

To help answer these questions let's look at an example of encrypting a text character with the Vernam cipher. Let's do the letter capital **A**.

The first step is to find the binary value of capital **A**. If we look **A** up in the ASCII table, we see it has a decimal value of 65 or a hex value of 41_{16}. I'm going to use the hex value to convert to binary because as I explained above it easier than trying to convert the decimal value. The first hex digit which is 4_{16}. Converting this to binary yields 0100_2. Next convert the second hex digit, which is 1_{16}, from hex to binary; and this is about as easy as it gets as it's 0001_2. So, the 8-digit binary value for capital **A** is $0100\ 0001_2$.

The next step is to get the bits for the key. Any set of 8-bits will do, and later in this section you will learn about generating strings of bits to use for keys and One Time Pads. But for now, we just need 8 bits so I'm going to pull some bits out of a hat and use: $0111\ 0000_2$.

Now that we have the plain text bits for **A**: $0100\ 0001_2$ and the one-time pad bits: $0111\ 0000_2$ we're ready to XOR them together to get the cipher text bits.

plain text bits (A):	`0100 0001`
key bits:	`0111 0000`
XORed bits:	`0011 0001`

To check the security of the Vernam cipher let's assume that you're an unauthorized user and you somehow discovered that the cipher text bits were 0011 0001. Is there any way for you to work backwards discover the plain text bits? If you look at any of the cipher text bits, and then look at the XOR table, you will see that the plain text bit could always be 0 or 1. So the plain text bits could be any 8 bit binary number! The only way to get the plain text bits from the cipher text bits is if you know the key bits.

When we encipher bits with the Vernam cipher the result is going to look like nonsense, just like what happens when text characters are enciphered. Even though the XORed bits may correspond to some ASCII character, in the case of our example $0011\ 0001_2$ is the ASCII for the number **1**, this doesn't really matter. What matters is that we could send the enciphered bits to someone else, and they would only be able to get back to the bits 0100 0001, or the ASCII character capital **A**, if they knew the bits from the key.

Before you move on, let me just empathize with you for a minute. When I was starting out in cryptography, I found that using XOR and the Vernam cipher were concepts that seemed to be both simple and complicated at the same time. Combining two bits with XOR seemed super simple, but I couldn't see how XORing bits was anything like using the Vigenère table. Hopefully you already see this, but if you're like I was and it's not clear at this point, then I suggest you take a little more time and go over the explanation again, or look at some other resources until it makes sense to you.

One-Time Pads with Bits, and Randomness

The next building block you will learn about for digital cryptography is generating random numbers or more specifically random bits to use to act as a One-Time Pad (OTP) in Vernam ciphers. Remember from text-based cryptography that the OTP provides perfect security, but only if the text in the key phrase does not repeat or follow a predictable pattern. The same holds true for digital cryptography, except the OTP needs to be a string of bits instead of a string of characters. Perfect security can be achieved if the OTP is a set of bits which are as long as the bits being encrypted, and the OTP bits don’t follow a pattern.

But, just like with text-based cryptography, generating and distributing the one-time pads introduces additional problems. Remember that manually generating truly random strings of characters for text-based one-time pads is a problem. It isn’t hard to generate one single short string of text, but it’s very difficult to continuously generate truly random longer strings. And once the random strings are generated there’s still the problem of sharing the pad between the sender and the recipient. These might seem like easy problems to solve, but they’re deceptively difficult, for both text-based and computer based digital cryptography.

For digital cryptography generating the random bits for the OTP is done with something called a Random Number Generator (RNG). If you’ve done any programming, you know that most programming languages have a built-in function that will return random numbers, but as you’ll learn these built-in functions may or not be sufficient for cryptography. Many of these built-in functions weren't designed with cryptography in mind and the random numbers they return end up repeating in patterns that would make any encryption system that used the random numbers easy to break.

So, how do we meet this challenge and generate a truly random set of bits that are as long as the message we want to encrypt, and that don’t repeat or follow a pattern? In this section you will learn about some of the difficulties in generating random binary OTPs, and how it is accomplished.

Introduction to Random Numbers

Random numbers are used for lots of things, casinos use them with gambling machines, scientists use them to model events that require actions to occur at random or semi-random intervals, and cryptographers need random numbers or random bits to build things like one-time pads. It might seem like it should be easy to generate random numbers, but surprisingly when dealing with computers, even modern computers, it turns out the opposite is true as it’s difficult to create truly random numbers. The first step in understanding the problem is to get an idea of what true randomness is or what it looks like, and how hard it is for humans to act randomly. The next step in finding a solution is to learn about the two classes of random numbers and random events, true or truly random numbers, and pseudo random numbers. After you learn about the difference between true random numbers and pseudo random numbers you will learn about the methods for generating pseudo random numbers.

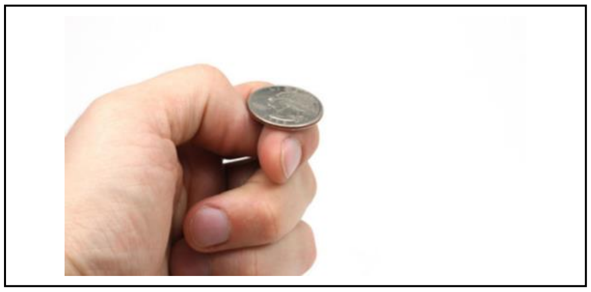
What is Randomness

A critical factor in understanding random numbers is to understand what randomness really means. The following two web sites may be older, but they have pretty good explanations of what it means to be random, and how hard it is for humans to generate random data manually. You don’t need to memorize the statistics from these pages, just make sure that you understand the main concept of randomness. If these links no longer function try doing your own search.

http://faculty.rhodes.edu/wetzel/random/mainbody.html - This page by Dr Christopher Wetzel contains information about randomness plus some great exercises that allow you to measure your ability at acting in a random fashion. And by acting random I don’t mean acting in the funny unexpected random way, I mean acting in a statistically random way.

https://www.fourmilab.ch/rpkp/experiments/statistics.html - Introduction to Probability and Statistics. This page by John Walker provides an easy to read introduction into the math behind the statistics behind randomness. Once again, please don’t worry about understanding every detail of the math, just try and understand the general concepts.

Truly or True Random Numbers



When we think of random numbers most people think of truly random numbers, like we get from the classic coin flipping example, where you flip a coin and if it comes up heads you get a 1, and if it comes up tails you get a 0. This could be used to generate a string of bits, by flipping the coin to obtain a 0 or 1, and flipping the coin again any time another bit is needed. Random numbers generated in this fashion are called true random numbers or truly random numbers because it's impossible to predict what the next number will be, and the numbers generated never follow a pattern. The technical term for never following a pattern is aperiodic. If something is periodic it means that it will repeat itself over some given period of time. True random numbers are also non-deterministic, which means given any number in a sequence, there's no way to determine the next number.

Another example is to use a six sided die to generate numbers. Each number has 1-in-6 chance of coming up at any time, so any sequence of numbers has an equal chance of being generated at any time. That is, some sequence like 2-1-5 will not appear any more than any other 3 rolls. There's also the often confusing statistical corollary that this sequence is just as likely to be rolled at any time as any other sequence. The main point is that numbers and sequences generated by rolling a six sided die will be totally unpredictable. A consequence of this is that it will be impossible to recreate or duplicate a sequence at a later time. This can be a drawback to using truly random numbers if you need to recreate the same sequence. I know this is super basic, but we're just locking down some characteristics of truly random numbers so we can compare them with pseudo random numbers.

Another characteristic that is important to consider for cryptography is how the random numbers are generated. Generating truly random numbers is typically done by measuring a naturally occurring process with lots of entropy like radioactive decay or doing things like flipping a coin or rolling dice. There are even some very cool systems like the one developed by a company named Cloudflare that generates random numbers by taking pictures of a wall of lava lamps and then digitizing the image data. Any type of system that generates truly random numbers is called non-deterministic because there's no way to determine in advance what the next result will be.

Some organizations such as Random.org[6] and Fourmilab[7] in Switzerland have implemented systems that generate true random numbers and will supply you with a set of truly random bits. Random.org has a web site that will generate numbers (bytes) based on atmospheric noise. You can go to their site at any time and generate up to 16,384 bytes of random data. The Fourmilab site in Switzerland generates random numbers based on radioactive decay. If you want a set of random numbers you have to fill out a form, but this is much simpler than building your own radioactive particle counter. While it's awesome that Random.org and Fourmilab will provide you with truly random bits, using them in a process to encrypt data would be slow and cumbersome. Each time you wanted to encrypt data you'd have to go to their web site and request more bits.

Generating truly random numbers by using a physical action like flipping a coin or rolling a dice is ok if you don't need many numbers, but it doesn't work very well for systems that require lots of random numbers. It's also difficult to use random numbers generated using one of these methods with a computer as it requires special hardware to measure the physical event and turn it into a random number, and then interface the random number generator with the computer. Plus, if you want to use the random numbers for encrypting and decrypting, you would also need to transmit the random bits you use as the encryption key to the recipient so they could decrypt the message. Any attacker that intercepts the encrypted message would also be able to intercept the key bits, and thus read the message.

[6] https://www.random.org/bytes/
[7] https://www.fourmilab.ch/hotbits/secure_generate.html

Pseudo Random Numbers

The problems with using true random bits were severe enough that it led cryptographers to search for ways to get computers to generate random numbers or random bits. Random numbers that generated by a computer are called pseudo random numbers because pseudo means something that superficially appears to be the real thing, but on closer inspection turns out to not be genuine. In most examples this has a negative connotation, like pseudo-leather , pseudo-science or pseudo-intellectual. But in the case of pseudo random numbers it's more used as an identifier to distinguish them from truly random numbers. Pseudo random numbers aren't inherently bad, in fact they're preferred over true random numbers in many situations, but they should not be mistaken for truly random numbers.

While true random numbers are generated by measuring some phenomena in the real world like coin flipping, pseudo random numbers are typically generated by running a computer algorithm called a Pseudo Random Number Generators (PRNG). The PRNG algorithms attempt to produce results that are as random as coin flips, but they have to use some type of math calculation or algorithm to produce random numbers. This means the results end up being not truly random if you look at them close enough or look at extremely long sequences of output. Even though PRNGs can't produce truly random numbers, there are a few that are well suited for use in digital cryptography.

Here are three ways that PRNGs and pseudo random numbers differ from true random numbers:

1. The first difference is that PRNGs can generate random numbers very quickly. This is because the algorithms used to generate the pseudo random number strings are relatively simple, and the programs written to implement the algorithms can run very quickly. This process is typically much, much faster than measuring some physical phenomena and then turning that data into a random number.

2. The second difference is that PRNGs and the numbers they produce are deterministic, or in other words that it's possible to determine, and more importantly control the random number sequences that are generated. This makes it possible to reproduce the same random number sequence at any time. While this might seem like a bad characteristic it's actually a desired trait in many situations like scientific simulations that require some

random input, but also need that input to be reproducible at a later time. For example, aeronautical engineers may want to model the stress on airplane wings. To do this they need to know the passenger and baggage weight, air temperature, fuel load etc. but this will vary from flight to flight and from minute to minute. Rather than running the simulation one time using a single average number for each of the values, they can run the simulation multiple times using numbers for the variable weight, temperature, fuel, etc. that are random but fall within a statistically accurate range. The aggregate results from running the simulation multiple times should be more accurate that a result obtained by running the simulation once. The important point is if they do run the simulation multiple times and want to later reproduce the same results, they will need to use the same random numbers.

Another place that using deterministic PRNGs is important is in cryptology. As you'll soon see, if we encrypt a message using a string of random digits as a one-time pad, we'll need to recreate that same string of random digits to decrypt the message. Hopefully you can see that you can't just send the random bits in the OTP along with the encrypted message. If this was done, anyone that intercepted the message would be able to easily decipher it. Luckily, we have pseudo random numbers to accomplish this because if we were generating the one time pad using true random numbers there would be no way for the recipient, or the sender, to recreate the one time pad. PRNGs have something called a *seed* which controls the sequence of bits they produce. If you give the PRNG the same seed twice, it will produce the same sequence of bits. But if you start the PRNG twice, using different seeds, the sequence of bits produced will be different. This dramatically reduces the amount of information the sender and recipient must exchange to allow the recipient to decrypt a message. Instead of having to exchange the all the bits in the OTP, they can simply exchange the seed. The recipient can then reproduce the bits in the OTP by running the PRNG using the seed sent with the message.

3. The last major general difference between truly and pseudo random numbers is that the pseudo random numbers are *periodic* which means they have sequences or patterns of numbers that will eventually repeat. This is a weakness, and if the repetition occurs too quickly the PRNG will be worthless for use in cryptography. Remember that with OTPs messages can be cracked if the key length is shorter than the message. The time it

takes the pattern(s) to appear varies with the PRNG algorithm, and even the choice of seeds can affect when the sequence begins to eventually repeat. However, modern algorithms are able to produce numbers with periods so long that it's not an issue in most practical circumstances. That being said, there are a class of algorithms that are known as Cryptographically Secure Pseudo Random Number Generators (CSPRNG) because they are known to have periods that are extremely long. The CSPRNGs are a subset of the PRNGs.

General Description of PRNG Algorithms

Now that you know what a PRNG is, let's take a look under the hood and see how some simple PRNGs work. You'll first learn how they use a seed to control the generation process, then learn the details and math behind two PRNGs, the Middle Square Method and the Linear Congruential Generator (LCG).

How do PRNGs generate random numbers that are reproducible? The PRNG algorithms typically use a starting value called a seed to begin their calculations. Here's a super simple example of how a seed works using an algorithm that isn't an effective PRNG, but it is simple enough to illustrate how a seed functions. Say we use the following process to generate a sequence of numbers:

1. Read an initial value, the seed, from the user. Call this value X.
2. Generate the first number in the sequence by multiplying X by X-1.
3. Generate each following number in the sequence by incrementing X by 1, then multiplying the new X by X-1.

Walking through the process using a seed of 5 produces the following:

When X = 5 the number generated is 5*4 = 20
Increment X to 6, and the number generated will be 6*5 = 30
Increment X to 7, and the number generated will be 7*6 = 42
Increment X to 8, and the number generated will be 8*7 = 56
Increment X to 9, and the number generated will be 9*8 = 72

If you start the process with the number 5 it will produce the numbers: 20 30 42 56 72. If you start the process with a different seed, it will produce a different sequence. But every time you start the process with a seed of 5 it will reproduce the same sequence. Note that sometimes the seed is called a salt or a key. This can be confusing because in cryptography we use the term “key” to refer to something completely different. If you see references to the key, make sure that you understand the context in which it’s being used.

Let’s go back and look at the fact that PRNGs are deterministic. When we say pseudo random numbers are called *deterministic* this is really a shorthand way of saying they’re generated by an algorithm or formula. That is, given a formula and a seed, it’s possible to determine what every number generated is going to be by just performing the calculations. This makes it possible to regenerate the same set of random number any time we want, or given one number, we can determine what the next number will be. True random numbers are called *non-deterministic*, because it’s impossible to ever regenerate the same string of numbers, or given one number know what the next number will be.

PRNG Algorithms Details

Here are details of two older PRNG algorithms, the middle square method and the linear congruential generator (LCG). The point of looking at these algorithms is to get a general idea of how PRNG algorithms function and how the seed is used to initialize the process so it can be repeated. These two algorithms are not adequate for use in cryptography because they have some flaws. We’re just looking at them because the algorithms they use are relatively simple. While the math used isn’t complicated, it might be hard to understand with just a quick read through. But keep in mind it’s not critical that you memorize the details of how either of these algorithms work, you just want to get a general idea of how they function.

Middle Square Method

The Middle Square Method[8] is a PRNG that was designed by John von Neumann, one of the fathers of computer science. Note that we’re just looking at this algorithm because it’s one of the first that was developed and it’s pretty easy to understand, and simple to work through by hand. It’s not a good candidate PRNG for cryptography.

[8] https://hbfs.wordpress.com/2017/11/21/the-middle-square-method-generating-random-sequences-viii/

This algorithm works by taking a seed number that's at least 4 digits and squaring it. The middle four digits of the result become the first four digits of the pseudo random number. The process is then repeated by squaring these new 4 digits and taking the middle 4 digits of the new result. For example, if the seed is 2678 then first step to finding the next number is to calculate 2678^2, which is 7171684. The middle 4 digits of this are 7168, so that would be the first 4 digits of the pseudo random number. This process is repeated to find the next number, and squaring 7168 results in 51380224, so the next 4 digits of the pseudo random number are 3802.

$2678^2 \rightarrow 7171684 \rightarrow 7168^2 \rightarrow 51380224 \rightarrow 3802^2 \rightarrow 14455204 \rightarrow 4552$

This method may seem like it does a good job generating random numbers, but it's never used for cryptography because for some seeds it produces repeating sequences very quickly. For example, if you use a seed of 1000 the sequence is:

$1000^2 \rightarrow 1000000 \rightarrow 0000^2 \rightarrow 0000 \rightarrow 0000^2$

Linear Congruential Generator (LCG)

The second simple PRNG method we'll look at is called the Linear Congruential Generator[9] (LCG). This was used as a random number generator by both Windows and some versions of UNIX for many years because it's easy to implement and in some cases can run very quickly. The math for this PRNG is slightly more complicated than the middle square algorithm as it uses the MOD or remainder function, but it's still relatively simple and easy to work through by hand.

The general formula for the LCG is:

$$X_{n+1} = ((a * X_n)+c) \text{ MOD } 2^m$$

The values for a, c and m are predefined constants, and X_n is the seed. For any implementation the values a, c and m must be defined. Once defined they will always stay fixed, which is why they're called constants. Once the values for a, c and m are set the algorithm is ready to go to work generating numbers. To calculate the first random number, you plug in the seed value for

[9] https://aaronschlegel.me/linear-congruential-generator-r.html

X_n and perform the calculations: the multiplication, the addition and then the MOD function. The result of this is the first random number, but you also plug it back into next iteration of the formula as the new X_n to calculate the next random number.

The MOD Function

The last step in this algorithm is to apply the modulus or MOD function. The modulus function is also called the remainder function because that's exactly what it calculates and returns. That is if you divide any number by another number the modulus function returns just the whole number remainder. For example, if you divide 16 by 5 the full answer is 3 ⅕. But if we look at the modulus or full number remainder by calculating 16 MOD 5 the result is 1. That is, we don’t worry about how many times 5 goes into 16, we just want to know what the whole number remainder is. So, after figuring that the largest multiple of 5 that will go into 16 is 15, the modulus or remainder is calculated by 16 – 15, which results in 1.

Here are several more examples:

17 MOD 5 = 2
18 MOD 5 = 3
19 MOD 5 = 4
20 MOD 5 = 0.

For smaller numbers the remainder is simple to calculate by hand. For larger numbers like 79827345572354 MOD 87634 the process is more difficult just because the numbers are large. But the process is simple, do one division and one subtraction, so it won’t be difficult to perform MOD calculations if you use a calculator as long as you understand what its doing..

When you use the MOD function as it’s used in the LCG algorithm, it helps with the basic process of changing the number during each iteration so there will be some randomness between subsequent numbers. But another thing that the MOD does, and why it’s often used, is that is acts as a limiter. That is, any time the results of any calculation are run through the MOD function it will limit the results so they will be between 0 and the number your MODing by -1. For example, if you calculate the whole number remainder for any number you divide by say 47, the smallest the remainder can possibly be is 0. This occurs when you divide any number that’s a multiple of 47 by 47. That is if you divide 47/47 the remainder is 0. If you divided 94 (which is 47*2) by 47 the remainder is also 0. And on the other end of the range, when you MOD by 47 the largest remainder you will ever get is 46. For example this occurs when you calculate 46 MOD 47. That is, if you divide 46 by 47 the remainder will be 46, or 47-1. The result of MODing any number by 47 can never be larger than 46, otherwise you can divide another 47 out of it. So, any result you get by MODing a number by 47 is going to be in the range 0-46. You can apply this logic to MODing by any number and see that MODing by a number **n** will restrict the output to numbers in the set 0 through **n**-1.

Sometimes when I first see equations like the LCG equation I get a little intimidated and confused. But when I remember that the MOD function is being used to limit the results, so they end up in a specified range, the equation makes more sense. In the case of the LCG equation, the formula takes the seed number, multiplies it by one constant and then adds another constant. Addition and subtraction are both simple math operations, that will make the number bigger. The last thing that happens in the LCG is the MOD function is used to ensure that the returned values are always going to be between 0 and 2^m-1 which keeps the numbers from growing too large.

The BSD rand( ) function

Next let's look a specific implementation of the LCG which is BSD UNIX built-in function called `rand()`. The `rand()` uses the LCG algorithm with these values for *a*, *c* and *m*:

$a = 1103515245$

$c = 12345$

$m = 31$

Plugging these constants into the LCG formula results in the following equation:

$$X_{n+1} = ((1103515245 * X_n) + 12345) \text{ MOD } 2^{31}$$

Notice that with m=31 the MOD function becomes MOD2^{31}. This ensures the values returned by this version of the LCG will be between 0 and 2^{31}, or between 0 and 2,147,483,648.

Let's walk through generating random numbers with `rand()`. As an example let's assume we start with a seed of 17, which means the first calculation would be:

$$X_{n+1} = ((1103515245 * \mathbf{17}) + 12345) \text{ MOD } 2^{31}$$

The first thing that happens is 17 is multiplied by the *a* constant, which in `rand()` is set to 1103515245. This result of this multiplication is 18,759,759,165. This is added to the c constant, which in this case 12345. The result of this addition is 18,759,771,510. The last operation is the MOD, which checks for the remainder when 18,759,771,510 is divided by 2^{31} or 2,147,483,648. In this case the result of the MOD and the initial pseudo random value is 1,579,902,326. And no, I didn't do that MOD calculation in my head.

To calculate the second random number, plug the first random number, 1,579,902,326, back into the LCG `rand()` formula. The formula for calculating the second pseudo random number then becomes:

$$X_{n+1} = ((1103515245 * \mathbf{1579902326}) + 12345) \text{ MOD } 2^{31}$$

This process repeats as long as you want, or until it generates the desired number of random digits. As explained earlier the MOD function always limits the output to a certain range, in this case 0 - 2^{31} or between 0 - 2,147,483,648.

Here are the first five random numbers that would be returned using seeds of 17, 18 and 20759:

Iteration	Seed = 17	Seed = 18	Seed = 20759
1	1579902326	2147483648	664910084
2	1084774263	55243232	1129524973
3	794669028	802909337	2063950114
4	617843789	316279646	12345
5	861954818	520203071	1406932606

This shows that using that, at least upon a quick superficial inspection, the numbers generated by `rand()` appear random. It also shows that using different seeds results in random number sequences that vary from sequences started by other seeds. The last, and most important thing to note is that if you run the algorithm with a seed of 17 it will always return the same sequence shown in the table. Or if you start `rand()` with a seed of 18 it will always return the sequence shown for the seed of 18. So, if Alice generated a sequence of numbers, and then wanted Bob to generate the same sequence, all she would need to do is pass him the seed.

Windows rand() function

Windows also has a built in function called `rand()` which returns a sequence of pseudo random numbers. It's also based on the LCG algorithm, but Microsoft uses a different set of values for *a*, *c* and *m.* The Windows implementation sets these constants as follows:

$a = 214013$

$c = 2531011$

$m = 31$

So, the formula used is very similar to the BSD formula, as it looks like this:

$$X_{n+1} = ((214013 * X_n) + 2531011) \text{ MOD } 2^{31}$$

Windows adds one slight twist, where they take the calculated value, X_{n+1}, and divide by 2^{16} or 65536 before outputting the pseudo random digits. They keep the calculated value, X_{n+1},for the next iteration, but for some reason they do this division to the pseudo random digits which has the effect of making the pseudo random number smaller, and the range of possible results smaller as well. In this case the pseudo random numbers will always be in the range 0 – (2^{31}/65536) or between 0 – 32,768.

Assuming you start with a seed of 17, the first calculation would be:

$$X_{n+1} = ((214013 * \mathbf{17}) + 2531011) \text{ MOD } 2^{31}$$

This results in 6,169,232 which will be the value used in the next calculation. However, the pseudo random digits output will be (6,169,232 / 65536) which ~= 94. As you can see this reduces the number of pseudo digits output by the function from 7 to 2.

The formula for calculating the second pseudo random number then becomes:

$$X_{n+1} = ((214013 * \mathbf{6169232}) + 2531011) \text{ MOD } 2^{31}$$

Here are the first five random numbers that would be returned using seeds of 17, 18 and 20759:

Iteration	Seed = 17	Seed = 18	Seed = 20759
1	94	97	2293
2	26602	4583	16576
3	39	15113	28118
4	7720	9593	17375
5	21238	30678	22994

As you probably expected the results returned by the Windows version of `rand()` are much different than those returned by the BSD UNIX version. Most notably, the numbers are all much smaller since the Windows version divides each value by 65536 as the last step in the calculations.

Before we finish this discussion about the BSD and Windows implementations of the rand() function let me reemphasize that the purpose for showing you these details about the LCG algorithm isn't because there's an expectation that you will memorize the LCG algorithm, or memorize the constants used by BSD UNIX or Microsoft. The purpose is to give you an idea of how PRNGs use an algorithm that contains some basic math operations to generate pseudo random numbers, and how the starting seed will control what numbers are output.

Random Numbers vs. Random Bits

Most of the discussion about randomness, both in this paper and in most documents, is focused on numbers. This may be slightly confusing because Vernam cipher requires random bits, not numbers. While this may seem like an issue, it's really not as there are two main ways of moving from random numbers to random bits. You can probably guess that one method just uses the binary representation of each number as the bits. For example, if we were just using 1 byte for each integer number and storing the number as an unsigned integer, the number 5 would be 0000 0101.

The second method also uses the bits for an unsigned integer, but instead of using all the bits it just takes the high order bit or the low order bit. That is, it just takes the left most or right most bit and ignores the other 7 bits. This method is obviously less efficient than using all the bits, because it requires generating 8 numbers just to get 8 bits.

So even though random numbers and random bits are different things, I'm going to use the term random number in most cases throughout the rest of this section. If something absolutely requires or refers to random bits, I'll make sure and spell it out. But otherwise I'll just refer to random numbers.

Difficulty Evaluating PRNGs Using LCG as An Example

One other concept that we'll use the LCG algorithm to illustrate is how difficult it can be to determine whether a PRNG algorithm is acceptable for use in cryptography, or whether it has some flaws. While it's not obvious, the choice of the values for the of *a*, *c* and *m* constants in the LCG algorithm has a huge impact on the performance of the algorithm when it's actually implemented in computer code. Choosing a good set of values is apparently much trickier than it seems. For some values the algorithm works well, that is it runs quickly and returns values that are close to random. But for other choices of *a*, *c* and *m* it performs very poorly. Even with bad choices for *a*, *c* and *m* the algorithm can return random numbers that work ok for doing things like setting up card games or mahjongg layouts. But some sets of *a*, *c* and *m* are known to result in sequences that quickly devolve into patterns, which makes that implementation of LCG dangerous for cryptography. Like pretty much all of cryptography I personally wouldn't be able to look at the algorithm and tell you what a good or bad set of values of *a*, *c* and *m* would look like, but luckily there have been some super intelligent people who have already done the hard work for us.

The following web page shows the *a*, *c* and *m* values for random number generator functions in several different programming languages.
https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use

Depending on your math skills the LCG formula might seem complicated or it might seem relatively simple. However, when you get right down to essence of the formula it's just doing one multiplication, one addition and one remainder calculation to ensure we get a new number after each iteration. (The Windows version does the extra division to further limit the range of the result from 0 – 32,768). Even though the individual math operations are not too difficult, the problem with this, and most PRNGs, is that the combination of the math operations is convoluted enough that it's hard to determine whether it's going to return satisfactory pseudo random numbers or not. And by satisfactory we usually mean numbers that aren’t going to become cyclical or follow a pattern.

If the math were simple it would probably be easier to see that patterns would emerge. For example, assume the PRNG formula is:

$$X_{n+1} = (X_n + 1)$$

In this case it’s super simple to see that just adding one will generate numbers that follow an obvious sequence. The same is true if the algorithm is changed to do any single math operation such as one multiplication, one subtraction or one division. It’s also slightly more difficult if we combine two operations such as one multiplication and one addition. For example, if the algorithm is you can see how the random numbers will continue to grow in an obvious pattern:

$$X_{n+1} = (X_n * 10) + 4$$

Adding the remainder function in makes things a little more complicated, but if you start calculating remainder for small numbers it’s much easier to see what’s happening. For example, assume the PRNG formula is:

$$X_{n+1} = (X_n + 1) \text{ MOD } 2^2$$

In this case it’s relatively easy to do a few calculations by hand and see that since the last step algorithm MODs numbers by 4 it’s going to limit the numbers to be in the set 0, 1, 2, and 3. No matter what number you start with the numbers generated will very quickly fall into the pattern 0, 1, 2, 3. Of course this would be a horrible algorithm for generating random numbers, but the reason we’re looking at it is to give you an idea of what the MOD function does with numbers that are small enough to understand.

Changing the constants in the algorithm to smaller numbers is only meant to give you an idea of how the LCG generates numbers. Using larger numbers for the *a*, *c* and *m* constants will produce similar results, the generated numbers will just be a lot, lot larger which makes it hard for most people to just look at the algorithm and detect any obvious faults.

So, if you can’t look at an algorithm and determine whether it would be good at producing pseudo random numbers, how is this decision made?

One option for vetting LCG or any PRNG algorithm would be to run it for every possible seed value, and check to see if patterns emerge. The problems with doing this are that there are typically so many possible seed values that it would take a long time to run the test program for every possible seed. And even if you did this, identifying patterns in pseudo random numbers the program outputs would be extremely difficult.

In the case of the BSD and Windows implementations of the LCG algorithm there are 2^{31} possible seed values, or over 2.1 billion seeds. This would require running the test program over 2.1 billion times. For each run you would then have to inspect the output to see if the algorithm generated any patterns. Identifying patterns would be easy with a computer program, but only if you knew what pattern you were looking for. The patterns could be of any length, from two or three numbers up to patterns that are thousands or hundreds of thousands of numbers long. Plus, the patterns may not appear until the algorithm has run through hundreds or thousands of iterations. Imagine trying to write a program to identify every string of every length of numbers in a list that could be hundreds of thousands of numbers long, and then checking to see if that string is ever repeated. For example, can you see the pattern in the following string of numbers? There is a repeating pattern, it’s just very difficult to spot because its period is so long.

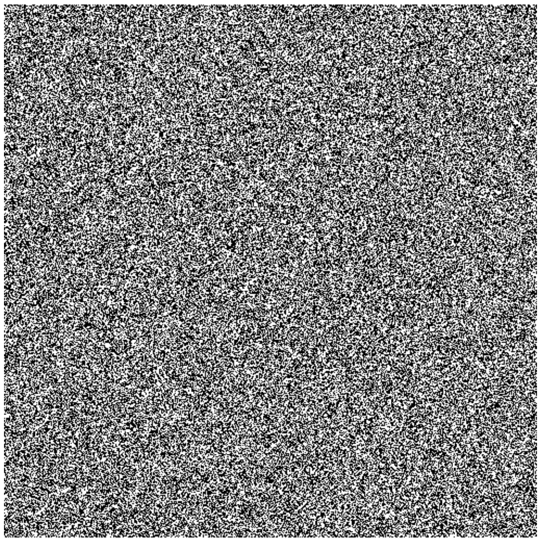
```
912712869219321375376502596327548512471024623642935703468235410236126351249186
231725318264102431875234102946127451285403472364293543540235704326512873642389
562783492837642364237488723540236402364582340982375047503467034861395642836402
193846540236520945712904658276340219378512356213650329865235682651243526349123
642837469823764892734682734682736498365475236423866876876234612312444531409248
008909989823523434234234234235453456346364485623642356826512435263491236428374
698237648927346827346827364983654752364238668768762346123124445314092480089099
898235234342342342342354534563463644856236423568265124352634912364283746982376
489273468273468273649836547523642386687687623461231244453140924800890998982352
343423423423423545345634636448562364
```

The following shows the same list of random numbers, just formatted a little differently to make the pattern more obvious.

912712869219321375376502596327548512471024623642935703468235410236126351249186
231725318264102431875234102946127451285403472364293543540235704326512873642389
562783492837642364237488723540236402364582340982375047503467034861395642836402
193846540236520945712904658276340219378512356213650329865
235682651243526349123642837469823764892734682734682736498365475236423866876876
23461231244453140924800890998982352343423423423423545345634636448562364
235682651243526349123642837469823764892734682734682736498365475236423866876876
23461231244453140924800890998982352343423423423423545345634636448562364
235682651243526349123642837469823764892734682734682736498365475236423866876876
23461231244453140924800890998982352343423423423423545345634636448562364

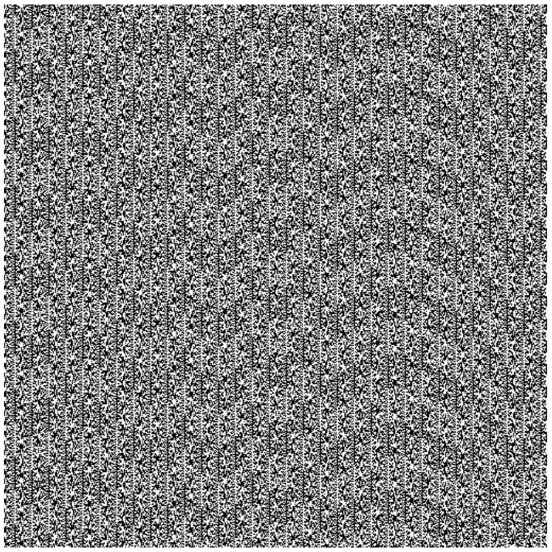
Luckily, once again there are some very intelligent people who have looked at this problem and come up with systems for evaluating PRNGs and their output. One of the most creative, and in my mind ingenious methods consists of creating an image by turning pixels white for bits set to 0 and black to bits set to 1. Here's an example of from a man named Bo Allen[10] who did this for true random data, and again for numbers generated by running code on Windows that uses the `rand()` function.

The true random bits:



The output from the LCG algorithm as it's implemented in Windows.

[10] https://boallen.com/random-numbers.html
https://www.random.org/analysis/



It’s easy to see the pattern in the numbers generated by the Windows implementation of LCG. The visual pattern indicates that the pseudo random numbers follow a pattern, which is why the Windows implementation of LCG should not be used for any application that requires a better approximation of true randomness. For example, you might consider using the Windows LCG for doing things like generating random landscape for a small game. But it should never be used for anything involving financial data or any type of sensitive data.

Another way to visualize the patterns generated by the LCG is to use the Random Walk program at the Khan Academy web page: https://www.khanacademy.org/computing/computer-science/cryptography/crypt/pi/random-walk-exploration

The random walk program displays two different paths, a white one that is generated by a random number function that receives a new seed in every step, and a blue one that is generated by a random number function that starts with a seed and continues to iterate. In both cases the PRNG algorithm used is called the Multiply-With-Carry (MWC) which is a form of an LCG invented by George Marsaglia[11]. With each step the program decides whether to grow in the North, East, South or West direction based on the random number.

To start the program, you must enter a seed that will be used for the blue path. Just click in the box and type your number. The cursor won’t move to the box when you click in it, so the user

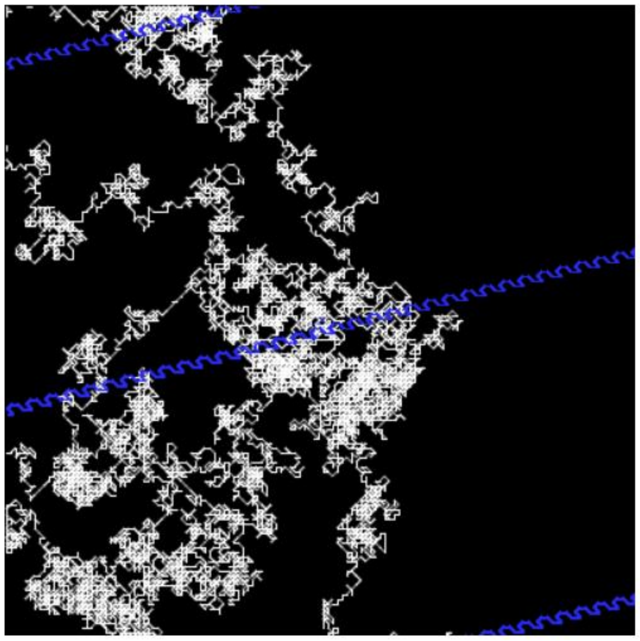
[11] https://en.wikipedia.org/wiki/Multiply-with-carry_pseudorandom_number_generator

interface is visually deceptive, but your seed number should display as you type it. Once you've entered your choice of seed click the Run button, which will start the random walk. If you want to restart the program you have to reload the page.

As the program runs both paths will start in the middle of the screen. The white path should wander in what seems to be a truly random pattern. It may even wander off the screen, but it usually returns if given enough time.

The blue path may also start out wandering in a random fashion, but sooner or later it will start to repeat in a visible pattern. The amount of time it takes to regress to the pattern depends on the seed. For example, given a seed of 11 causes the blue path to repeat a very small pattern right in the middle of the screen.

The following illustration shows the results of using a seed of 444. This causes the blue path to repeat in an obvious pattern.



The third option for evaluating PRNGs is to perform some statistical analysis on their output. These tests were developed to check the output from true random number generators like the site that measures radioactive decay or the site that measures atmospheric noise. These tests are described at one of the pages at Random.org https://www.random.org/analysis/

The last option for evaluating a PRNG is to hire a savant or genius. There are some super gifted people who can look at an algorithm and with a little thought identify weaknesses or flaws; but even they might not be able to spot every problem. This is why some algorithms, like the LCG, may be widely accepted and used for years, but replaced later on when a flaw is identified.

Cryptographically Secure Random Number Generators

If we summarize the information you've learned about random numbers so far, we have these main points:

1. For the Vernam cipher to be effective it needs random numbers that do not repeat in a predictable way.

2. True random numbers can be used for the Vernam cipher, but they are difficult to generate and there's no way to reproduce a sequence of true random numbers, which is necessary to decrypt messages.

3. PRNGs can generate sequences of numbers that approximate true random numbers, they are efficient, and they can reproduce a set of numbers as the sequence they generate is controlled by a seed.

4. Not all PRNG algorithms or implementations are able to produce sequences that are adequate for use in cryptography. The main disqualifying factor is if repeatable patterns are produced. It's difficult for most people to evaluate an algorithm and determine whether it will produce patterns or not.

That brings us up to the point where we need to find some PRNGs that are suited for cryptography. Luckily there are a set of PRNGs that are known as Cryptographically Secure Pseudo Random Number Generators (CSPRNG) that are available for use. They're called this because the numbers or bits they generate don't produce patterns, or if they do, the patterns have extremely long periods which means they don't appear for a long time.

Another important feature of CSPRNGs is that if you have knowledge of any number generated as part of a sequence, there's no way to determine the previous number. You can produce the following numbers, but you can't produce the previous numbers. This prevents anyone that intercepts a number in the sequence from reverse engineering all the previous numbers.

Here's a list of PRNGs that are also CSPRNGs:

1. Lagged Fibonacci Generator
2. Linear feedback shift registers
3. Blum Blum Shub

If you want to know the basics of these work, you can find information at the following web sites. If any of these sites are no longer available, you can always do your own search.

https://medium.com/asecuritysite-when-bob-met-alice/for-the-love-of-computing-the-lagged-fibonacci-generator-where-nature-meet-random-numbers-f9fb5bd6c237

https://pthree.org/2015/05/29/the-lagged-fibonacci-generator/

https://www.embeddedrelated.com/showarticle/1065.php

https://asecuritysite.com/encryption/blum

https://www.commonlounge.com/discussion/8275a95fbc0d4b53b4bec1fc72cee565 - Information on LCG and Blum Blum Shub

http://www-math.ucdenver.edu/~wcherowi/courses/m5410/m5410fsr.html

You'll also learn about several other CSPRNGs that are built into encryption algorithms later in the book. It's not critical that you understand the inner workings of any of the CSPRNG algorithms. In fact, it's not even suggested that you look at the math involved unless you're extremely experienced and literate in mathematics and number theory. You should however be aware of the names of these CSPRNGs, especially if you're planning on taking a certification exam.

You should be aware that we use the CSPRNG algorithms constantly as we use the web as they're built into things like SSL and TLS which have become part of the Internet infrastucture. You'll learn about this later in the book, but for now you should start considering a few points:

1. We need random numbers to keep web traffic secure.

2. There are PRNG algorithms that have been officially blessed or recommended for use as CSPRNGs.
3. Checking PRNG code to ensure it can function as a CSPRNG is extremely difficult, which means we have to trust the expert recommendations.

The main point, which you'll see in many aspects of modern cryptology, is that we need to trust the experts. This means we need to trust that they have only selected and approved algorithms that meet the necessary requirements to do a certain job, such as generating random numbers. But it also means that we have to trust that they didn't leave any backdoors which will allow them to read anyone's secure communications. History has shown that in both cases, we, meaning the public, have sometimes trusted organizations and algorithms that we should not have.

Of course, I'm in the same boat as you and the rest of the Internet users. I didn't check any of these CSPRNG algorithms myself. I'm taking reassurance from the mathematicians and cryptographic professionals that these are actually cryptographically secure.

How to implement or use a PRNG

Now that you have some idea of how random numbers are generated let's talk about applying this knowledge. There are all kinds of PRNG functions and programs you can use, and code you can download from GitHub or other repositories. And when this code is implemented you may need to do some configuration, such as setting constants for the PRNG algorithm, or selecting seeds. This takes some knowledge of how the PRNG algorithm functions, because as you've seen if poor constants are used or if the wrong seed is used it can lead to patterns in the random numbers, which can make it easy to break messages. But do you need to worry about this? Either as a user, or as a cyber security professional?

The answer depends on the situation. There are two general situations where you'll be using random numbers. The first is when you use programs that someone else wrote that use random numbers, and the second is if you're a programmer or developer writing your own program or application that needs random numbers.

Users – No Worries

In the first case, where you're using applications or services that are using random numbers, you probably don't have to worry about how the random numbers are generated if the standard security suites are being used. Most people fall into this category, the category of general users. And in fact, most people have no idea that random numbers are even involved as they make purchases on the Internet and exchange sensitive information via HTTPS, or encrypt email messages or files on their computers. As users we're lucky that every major OS includes suites of strong cryptographic functions which include battle tested CSPRNGS. Most users have been trained to check a web site for security assume the site is secure if they see it's using HTTPS and a lock icon. But the average user most likely doesn't even know they're using random numbers as they don't have to set any parameters, or seeds for the algorithms. And they have no idea what this means about the math involved to make the encryption secure.

General Application Programmers – Some Knowledge Required

The second group of people who use random numbers are the programmers and software engineers. We should subdivide them into two groups as well, those that write code used in security suites, and those that need random numbers for applications that don't deal with security. The programs that don't deal with security are things like games, where random numbers are used to shuffle cards or build world layouts, or applications that need random numbers to model real world variability for things like engineering or human behavior modelling. The programmers who are **not** working on security code need to have a decent working knowledge of random number generators and the functions that can be used to generate them, but they don't have to have the intimate knowledge required for anyone dealing with security.

For example, a game programmer can most likely use a random number generator that relies on entropy as they won't need to reproduce the same string of numbers again. They'll need to know enough to not use a PRNG that will always produce the same random numbers, otherwise their game won't have any variability. But anyone writing a program that needs to have results that can be reproduced needs to know enough to use a PRNG that will generate the same set of numbers if given the same seed.

Security Programmers – Expert Knowledge Required

If you're going to write any code or application that requires strong security, here is some important advice I strongly urge you to heed:

1. Do NOT write your own code. If you need random numbers for a secure application, then there’s a chance that you might consider trying to write your own code for encryption. This is extremely dangerous as making even the slightest mistake can cause vulnerabilities and make it easy for an attacker to read the encrypted data. There are encryption algorithms available in every mainstream OS and most programming languages which should be used instead. These readily available algorithms have been scrutinized and tested by experts in cryptography before being put into production and have withstood attacks from every angle in the years they have been in use. To see the risks of writing your own PRNG think of this analogy. Your company needs to fly people to various places around the globe and you’re tasked with finding an aircraft to do the job. There are several jets that you can use for free. Or you can build your own jet from scratch. If you decide to build your own jet you can find free blueprints and plans that show you how to build jet engines, the airframe, the control systems, everything you need to build a jet. But if you make a single mistake while you’re building your own jet the results could be deadly. If you forget to tighten a single bolt, or miss one wire on a control system, your jet might fly for a while, but fail unexpectedly with disastrous results. And, as you make the decision whether to use one of the free planes or build your own, remember that as the person who built the jet you will be held personally responsible for the safety of anyone flying in the company jet. Applying this analogy to coding means that if you decide to write your own PRNG you’ll be putting all your company’s data at risk. And if the data is ever compromised due to a flaw in your code you will be held personally responsible.

2. Do NOT write your own code. There are a few security related applications other than encryption that require random numbers. For example, maybe you need to generate random numbers for generating one time pass codes to text to users when they access their account from a new device, or for generating CAPTCHAs. In this case you should realize that every mainstream OS and programming language have random number generation functions available. Just as with the full encryption code it may be tempting to write you own programs or functions as the algorithms for the various random number generators are widely available. However, if you’ve done any programming at all you know it’s easy to make mistakes. So, the recommendation is to use one of the built-in functions instead of taking the risk of writing your own code.

3. Even if you decide to use the built-in cryptographic functions, you'll need a strong understanding about how the specific random number generation algorithm functions. When you implement the function, you may be required to provide configuration information such as the constants to be used in the main algorithm, seed value, or constraints on the output. Without a decent working knowledge of the algorithm it's possible to choose poor values that cause the algorithm to generate non-random sequences.

Random Number Summary

You've been presented with a lot of background information on random and pseudo random numbers, random bits and PRNGs. I know that the amount of information may seem a little overwhelming, so here's a summary of what I believe are the critical key points.

True random numbers are relatively difficult to generate as they typically require measuring occurrences of physical phenomena in the real world. This makes them inefficient for use in processes that require large amounts of random numbers. True random numbers are non-deterministic, which means that it's impossible to generate the same string of random numbers more than once. And finally, true random numbers are aperiodic, which means they will not have patterns or sequences of numbers that will repeat.

Pseudo random numbers are generated by algorithms called pseudo random number generators (PRNG) which are very efficient and can create long strings of pseudo random numbers very quickly. Pseudo random numbers are deterministic, which means that it's possible to recreate the same string or pseudo random digits. This is possible because the sequence generated by a PRNG is controlled by a starting seed. Being efficient and deterministic make pseudo random numbers better candidates for usage with cryptography than true random numbers. The one characteristic of PRNGs that is not desirable is that they generate strings of numbers that will eventually be periodic.

There is a class of PRNGs called CSPRNGs that are periodic, but the repetition takes a long time to develop, long enough that the problem can usually be ignored. CSPRNGs are used to produce the stream of random bits required as the key in the Vernam cipher.

Is There Actually Any True Randomness?

The last bit of randomness I'm going to leave you with is a philosophical question. This question is "Are there really random events, or is everything pre-determined?" Most humans believe that there is randomness at some levels in our lives, and that we have some choice regarding our actions. But several astrophysicists and cosmologists have presented arguments that everything that's happened after the big bang is pre-determined in an argument called super-determinism[12]. The argument is taken further by philosophers who note that we know the rules that determine how atomic and sub-atomic particles interact. That is, we know there are rules the govern every chemical reaction, and every electrical or magnetic reaction. And we know the rules for how very large objects move and interact, such as the movement of stars and galaxies. In other words, there are deterministic rules that govern every action and motion in two levels of physical reality, the level much smaller than human perspective and the level much larger than human perspective. If we agree that the way things interact at these two levels are predetermined, then wouldn't this mean that they're also predetermined at a human level? This is a big question in Philosophy, and a huge rabbit hole that you can travel down if you want. But in order to proceed in the class I'm going to ask you to assume that there are truly random events.

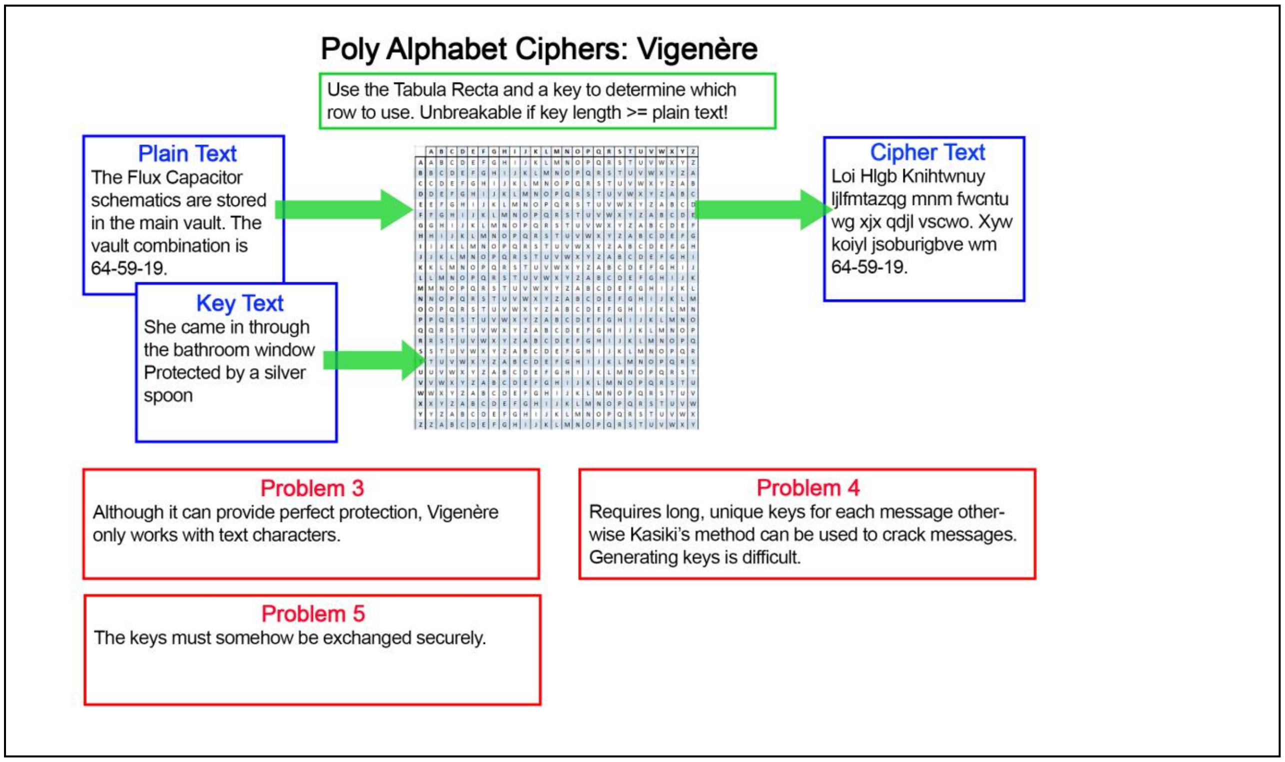
Chapter Summary

Wow! Does your brain feel full? I know mine does every time I go back over this material. Let's summarize the information in this chapter, starting by looking back at the main problem representing text characters as ASCII binary numbers, the XOR function, and random numbers are meant to solve.

In the last chapter you learned that the Vigenère cipher can provide perfect protection as long as the key text is as long as the plain text, and the keys are only used once. The system for generating and using keys in this way is generally referred to as a One Time Pad or OTP. Once again, this can provide perfect protection, but the entire process had a few issues. Note that the numbering for these issues is based on where they sit in the overall list of solutions and problems encountered in the evolution of modern cryptography.

[12] https://explore.scimednet.org/index.php/bells-theorem-is-everything-predetermined/

3. The Vigenère only works with plain text characters. No other type of data can be encrypted with this system.
4. Generating the One Time Pads is difficult.
5. Securely exchanging the One Time Pads is difficult.



Poly Alphabet Ciphers: Vigenère

Use the Tabula Recta and a key to determine which row to use. Unbreakable if key length >= plain text!

Plain Text
The Flux Capacitor schematics are stored in the main vault. The vault combination is 64-59-19.

Key Text
She came in through the bathroom window Protected by a silver spoon

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

Cipher Text
Loi Hlgb Knihtwnuy ljlfmtazqg mnm fwcntu wg xjx qdjl vscwo. Xyw koiyl jsoburigbve wm 64-59-19.

Problem 3
Although it can provide perfect protection, Vigenère only works with text characters.

Problem 4
Requires long, unique keys for each message other-wise Kasiki's method can be used to crack messages. Generating keys is difficult.

Problem 5
The keys must somehow be exchanged securely.

The solution for Problem 3, the fact the Vigenère cipher only works with text is solved by finding a way to make it work with every kind of digital information had two components. The first was learning how all digital information is stored as binary data, and how to convert text data to binary and vice-versa using the ASCII table.

Solution 3A

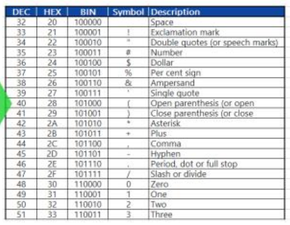
Represent all text and information using binary.

Digital Cryptography: ASCII Text

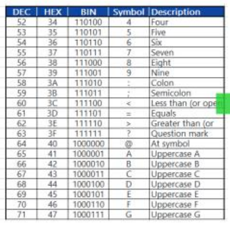
Use the ASCII table to convert text to binary. Use other digital formats for images, video, sound, etc.

Plain Text

The Flux Capacitor schematics are stored in the main vault. The vault combination is 64-59-19.



DEC	HEX	BIN	Symbol	Description
32	20	100000		Space
33	21	100001	!	Exclamation mark
34	22	100010	"	Double quotes (or speech marks)
35	23	100011	#	Number
36	24	100100	$	Dollar
37	25	100101	%	Per cent sign
38	26	100110	&	Ampersand
39	27	100111	'	Single quote
40	28	101000	(	Open parenthesis (or open
41	29	101001	)	Close parenthesis (or close
42	2A	101010	*	Asterisk
43	2B	101011	+	Plus
44	2C	101100	,	Comma
45	2D	101101	-	Hyphen
46	2E	101110	.	Period, dot or full stop
47	2F	101111	/	Slash or divide
48	30	110000	0	Zero
49	31	110001	1	One
50	32	110010	2	Two
51	33	110011	3	Three



DEC	HEX	BIN	Symbol	Description
52	34	110100	4	Four
53	35	110101	5	Five
54	36	110110	6	Six
55	37	110111	7	Seven
56	38	111000	8	Eight
57	39	111001	9	Nine
58	3A	111010	:	Colon
59	3B	111011	;	Semicolon
60	3C	111100	<	Less than (or ope
61	3D	111101	=	Equals
62	3E	111110	>	Greater than (or
63	3F	111111	?	Question mark
64	40	1000000	@	At symbol
65	41	1000001	A	Uppercase A
66	42	1000010	B	Uppercase B
67	43	1000011	C	Uppercase C
68	44	1000100	D	Uppercase D
69	45	1000101	E	Uppercase E
70	46	1000110	F	Uppercase F
71	47	1000111	G	Uppercase G

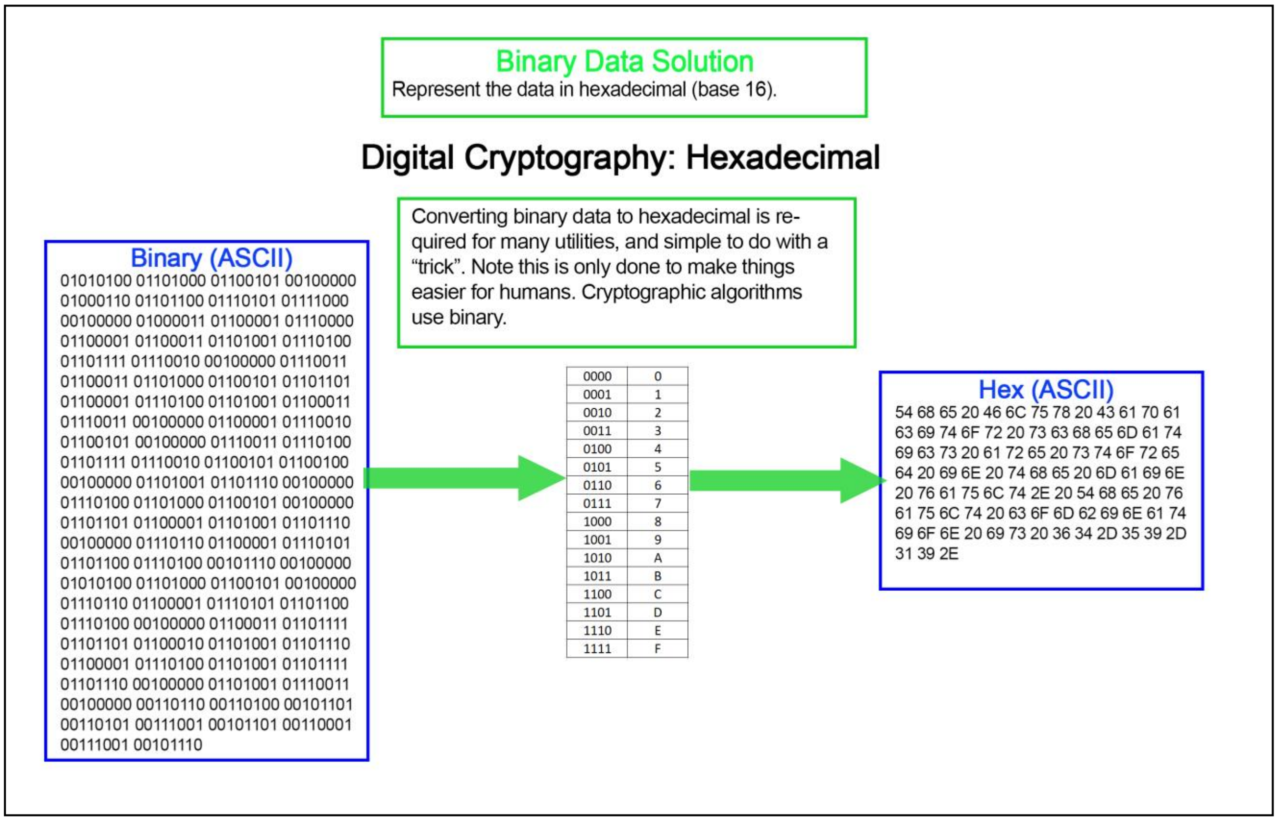
Binary (ASCII)

```
01010100 01101000 01100101 00100000
01000110 01101100 01110101 01111000
00100000 01000011 01100001 01110000
01100001 01100011 01101001 01110100
01101111 01110010 00100000 01110011
01100011 01101000 01100101 01101101
01100001 01110100 01101001 01100011
01110011 00100000 01100001 01110010
01100101 00100000 01110011 01110100
01101111 01110010 01100101 01100100
00100000 01101001 01101110 00100000
01110100 01101000 01100101 00100000
01101101 01100001 01101001 01101110
00100000 01110110 01100001 01110101
01101100 01110100 00101110 00100000
01010100 01101000 01100101 00100000
01110110 01100001 01110101 01101100
01110100 00100000 01100011 01101111
01101101 01100010 01101001 01101110
01100001 01110100 01101001 01101111
01101110 00100000 01101001 01110011
00100000 00110110 00110100 00101101
00110101 00111001 00101101 00110001
00111001 00101110
```

Binary Data Problem

Representing data in binary results in large amounts of data that are difficult for humans to read.

Because binary is cumbersome and hard for humans to read, you also learned how to convert text to hexadecimal, and convert between binary and hex.



Binary Data Solution

Represent the data in hexadecimal (base 16).

Digital Cryptography: Hexadecimal

Converting binary data to hexadecimal is required for many utilities, and simple to do with a "trick". Note this is only done to make things easier for humans. Cryptographic algorithms use binary.

Binary (ASCII)

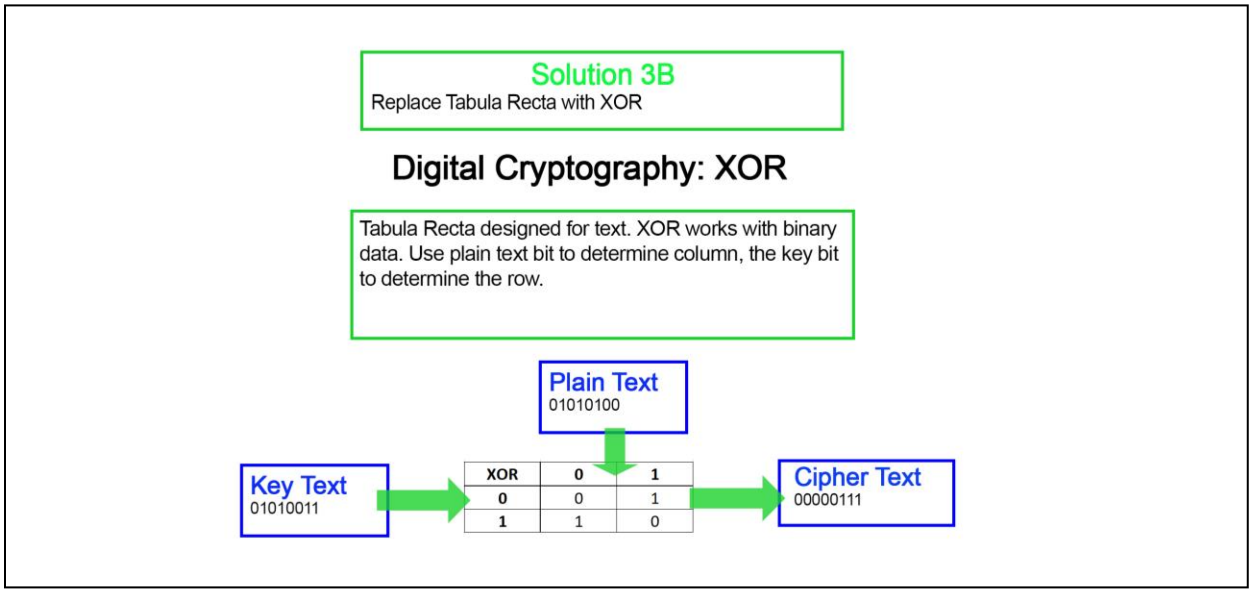
```
01010100 01101000 01100101 00100000
01000110 01101100 01110101 01111000
00100000 01000011 01100001 01110000
01100001 01100011 01101001 01110100
01101111 01110010 00100000 01110011
01100011 01101000 01100101 01101101
01100001 01110100 01101001 01100011
01110011 00100000 01100001 01110010
01100101 00100000 01110011 01110100
01101111 01110010 01100101 01100100
00100000 01101001 01101110 00100000
01110100 01101000 01100101 00100000
01101101 01100001 01101001 01101110
00100000 01110110 01100001 01110101
01101100 01110100 00101110 00100000
01010100 01101000 01100101 00100000
01110110 01100001 01110101 01101100
01110100 00100000 01100011 01101111
01101101 01100010 01101001 01101110
01100001 01110100 01101001 01101111
01101110 00100000 01101001 01110011
00100000 00110110 00110100 00101101
00110101 00111001 00101101 00110001
00111001 00101110
```

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

Hex (ASCII)

```
54 68 65 20 46 6C 75 78 20 43 61 70 61
63 69 74 6F 72 20 73 63 68 65 6D 61 74
69 63 73 20 61 72 65 20 73 74 6F 72 65
64 20 69 6E 20 74 68 65 20 6D 61 69 6E
20 76 61 75 6C 74 2E 20 54 68 65 20 76
61 75 6C 74 20 63 6F 6D 62 69 6E 61 74
69 6F 6E 20 69 73 20 36 34 2D 35 39 2D
31 39 2E
```

Once we had the text data in binary, the next component in the solution was finding a way to replace the Vigenère cipher and the tabula recta. The Vigenère cipher was replaced with the Vernam cipher which uses the XOR function instead of the tabula recta. You learned how to XOR bits, and how the XOR operation is used instead of the AND, OR, or any other Boolean operation because it's the only one that provides a unique mapping when decrypting.



Solution 3B

Replace Tabula Recta with XOR

Digital Cryptography: XOR

Tabula Recta designed for text. XOR works with binary data. Use plain text bit to determine column, the key bit to determine the row.

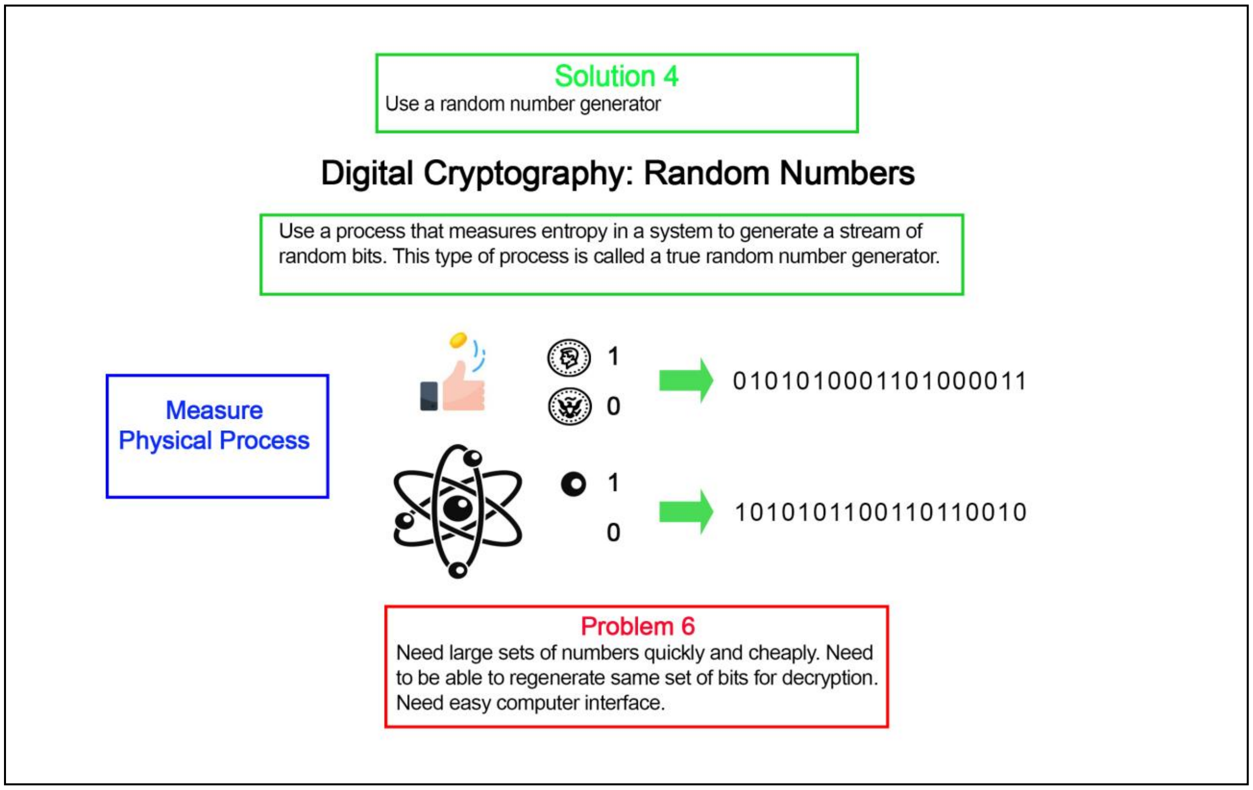
Plain Text 01010100

Key Text 01010011

XOR	0	1
0	0	1
1	1	0

Cipher Text 00000111

Using the Vernam cipher and the XOR table solved Problem 3, which brought us to finding a solution for Problem 4, which is generating a One Time Pad. In this case the OTP needs to be a string of random bits that can be XORed with the binary plain text data. The first step in this process was learning about true random numbers and true random number generators.



Solution 4

Use a random number generator

Digital Cryptography: Random Numbers

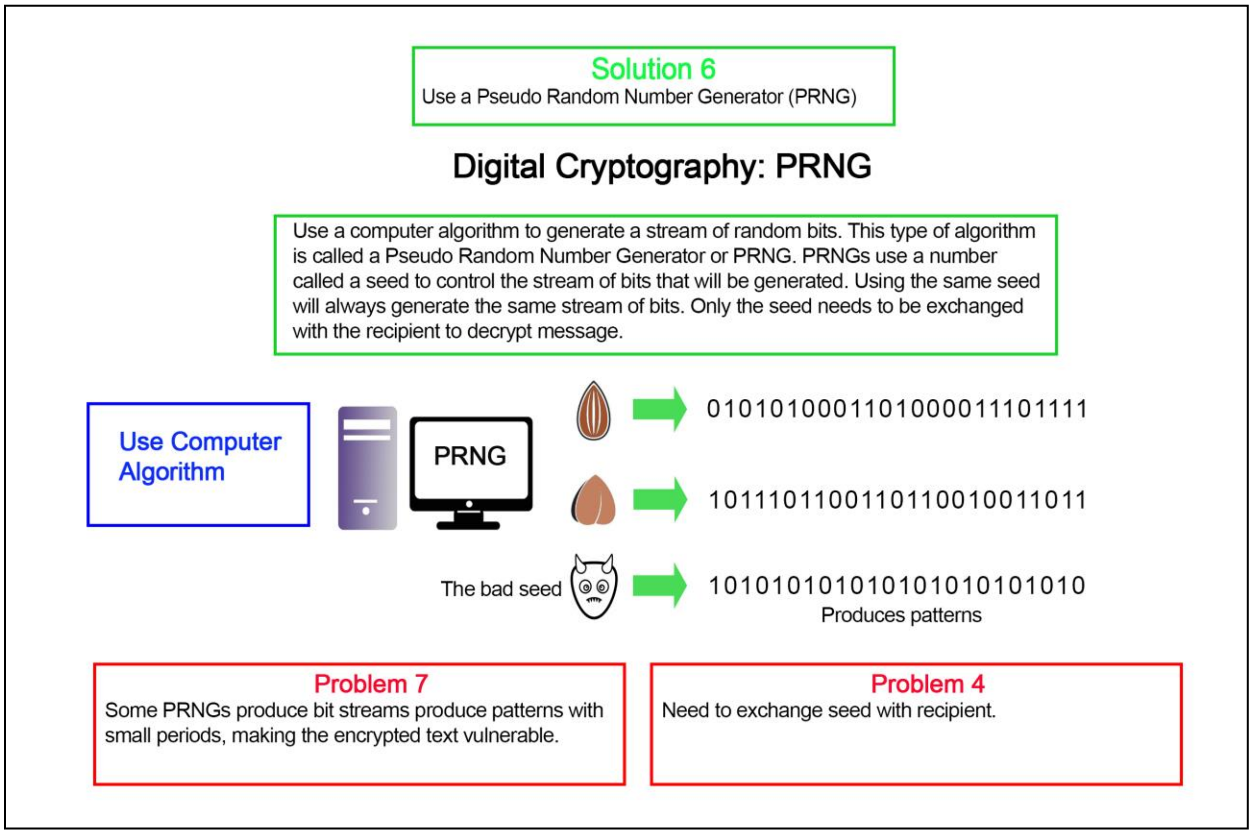
Use a process that measures entropy in a system to generate a stream of random bits. This type of process is called a true random number generator.

Problem 6

Need large sets of numbers quickly and cheaply. Need to be able to regenerate same set of bits for decryption. Need easy computer interface.

You also learned about Problem 6, which is how it's difficult to quickly obtain large sets of numbers, and how true random numbers are non-deterministic which makes them difficult to use for decryption.

Problem 6 was solved by using computer programs called Pseudo Random Number Generators (PRNG) to generate the OTP bits to use as encryption and decryption keys. You also learned about how PRNG algorithms are deterministic, which means they can reproduce the same set of bits repeatedly, and they use a seed to control which set of bits they will create. This eliminates the need for the sender and recipient to share the entire OTP. The only thing the recipient needs to decrypt a message is the seed.



Solution 6

Use a Pseudo Random Number Generator (PRNG)

Digital Cryptography: PRNG

Use a computer algorithm to generate a stream of random bits. This type of algorithm is called a Pseudo Random Number Generator or PRNG. PRNGs use a number called a seed to control the stream of bits that will be generated. Using the same seed will always generate the same stream of bits. Only the seed needs to be exchanged with the recipient to decrypt message.

Problem 7

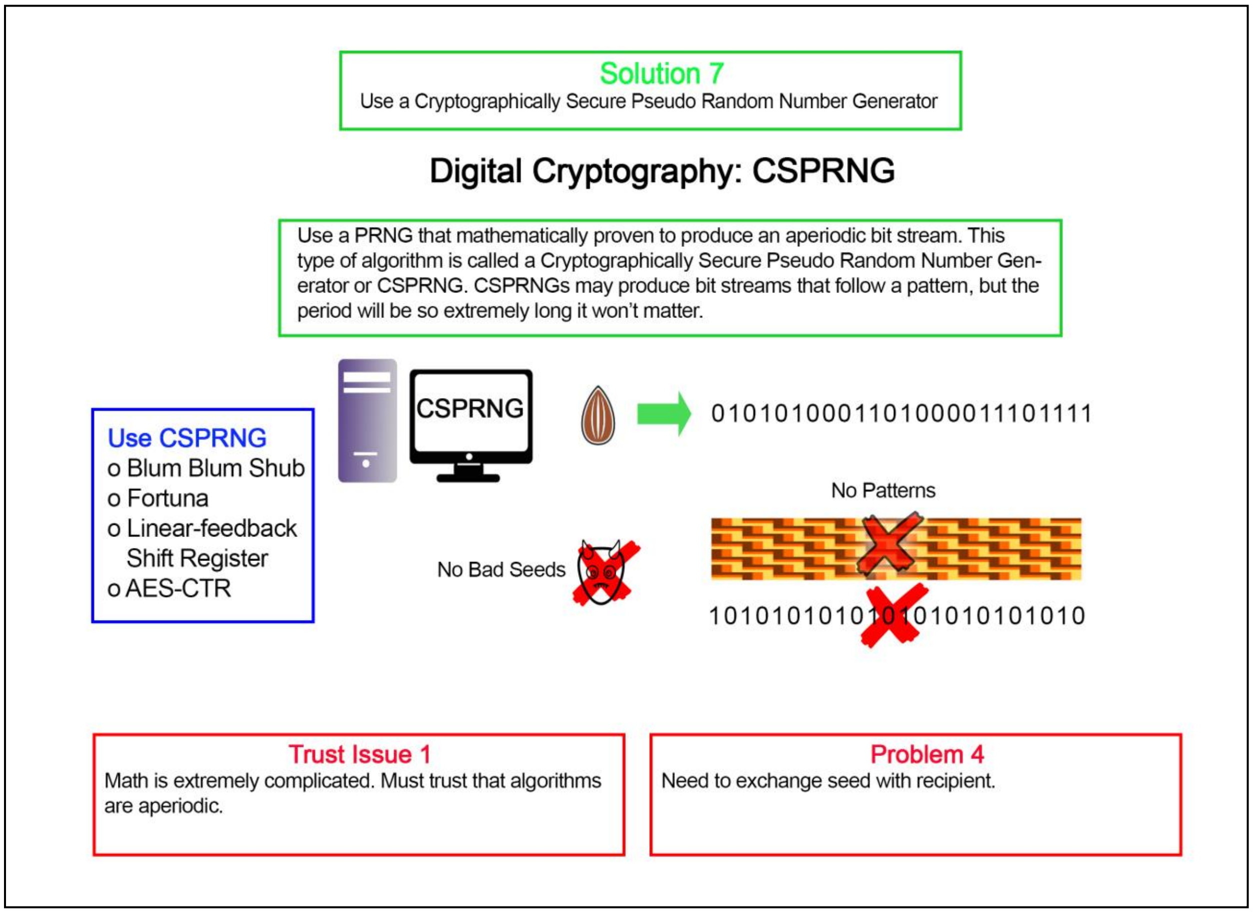
Some PRNGs produce bit streams produce patterns with small periods, making the encrypted text vulnerable.

Problem 4

Need to exchange seed with recipient.

While PRNGs are deterministic and provide an easy way to get random numbers on a computer, they also introduced Problem 7, which is that most PRNG algorithms will produce numbers that start to follow a pattern. When the PRNG numbers used for encryption follow a pattern the result is just like repeating a text key with the Vigenère cipher, it makes it easy to crack the encrypted text.

This problem was solved by finding a set of PRNGs known as Cryptographically Secure Random Number Generators or CSPRNGs. The CSPRNGs have been checked by experts and battle tested to ensure they do not produce numbers that follow a pattern, regardless of the seed.



Solution 7
Use a Cryptographically Secure Pseudo Random Number Generator

Digital Cryptography: CSPRNG

Use a PRNG that mathematically proven to produce an aperiodic bit stream. This type of algorithm is called a Cryptographically Secure Pseudo Random Number Generator or CSPRNG. CSPRNGs may produce bit streams that follow a pattern, but the period will be so extremely long it won't matter.

Use CSPRNG
- o Blum Blum Shub
- o Fortuna
- o Linear-feedback Shift Register
- o AES-CTR

Trust Issue 1
Math is extremely complicated. Must trust that algorithms are aperiodic.

Problem 4
Need to exchange seed with recipient.

Tying this all together, modern encryption uses something similar to the Vigenère cipher called the Vernam cipher, which works on binary data instead of text characters. The Vernam cipher uses the XOR function instead of the tabula recta and uses bits generated by a CSPRNG as the key. The only information the message recipient needs to regenerate the key bits is the seed for the CSPRNG. Finding a way to securely exchange the seed for the CSPRNG is a problem that must be solved.

In addition to the technical problem of the seed/key exchange, we now have a trust issue. The CSPRNG algorithms are so complicated that it's almost impossible to tell if they have technical issues or backdoors that would allow an attacker to regenerate the key bits without knowing the seed. This leaves us in the situation where we must trust the experts who have approved the current set of CSPRNGs.