5 Symmetric Cryptography: Stream Ciphers

Introduction

In the previous section you learned about the critical building blocks of a form of digital cryptography known as symmetric cryptography. In this section you will learn the definition of symmetric encryption, and the definition of the two main forms of symmetric encryption which are stream ciphers and block ciphers. You will also learn the details of two stream ciphers.

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

1. Describe the main difference between symmetric and asymmetric ciphers.
2. Describe the difference between how stream ciphers and block ciphers encrypt/decrypt text.
3. Describe in general terms how the RC4 and Salsa20/ChaCha20 PRNGs function.
4. List 2 weaknesses in the various implementations of RC4 that weaken it's ability to secure and protect encrypted messages.
5. Identify the specific weakness in WEP that made it vulnerable to compromise.
6. List the information that must be exchanged between the sender and recipient to use ChaCha20.
7. Identify which stream cipher is currently used in SSL/TLS.

Required Reading

1. **Read this document first as it will provide you with the framework regarding all the subjects covered in this section regarding symmetric ciphers, and stream ciphers.**

2. Here are some web sites you can use if you want to go further into any of the subjects in this chapter:

Basics of Symmetric ciphers
http://www.crypto-it.net/eng/symmetric/index.html
https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking

More information of SSL/TLS and WEP

https://www.websecurity.digicert.com/security-topics/what-is-ssl-tls-https
https://searchsecurity.techtarget.com/definition/Wired-Equivalent-Privacy
https://www.dummies.com/programming/networking/understanding-wep-weaknesses/

More details on RC4
http://www.rickwash.com/papers/stream.pdf - Deep dive on RC4 and stream ciphers
From https://stepuptocrypt.blogspot.com/2019/02/symmetric-cryptography-rc4-algorithm.html

Attacks on RC4
https://www.rc4nomore.com/ - More information on the RC4 attacks, and common sense counter measures
https://en.wikipedia.org/wiki/RC4 - section on attacks on RC4

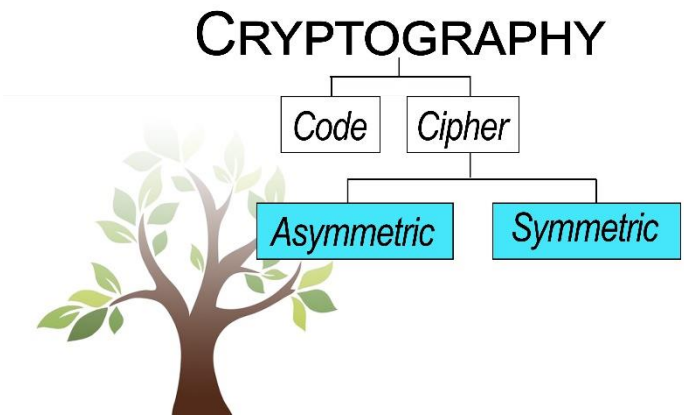
Salsa20 and ChaCha20

https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant – Wikipedia description of salsa and chacha
https://ianix.com/pub/chacha-deployment.html - software and systems using ChaCha

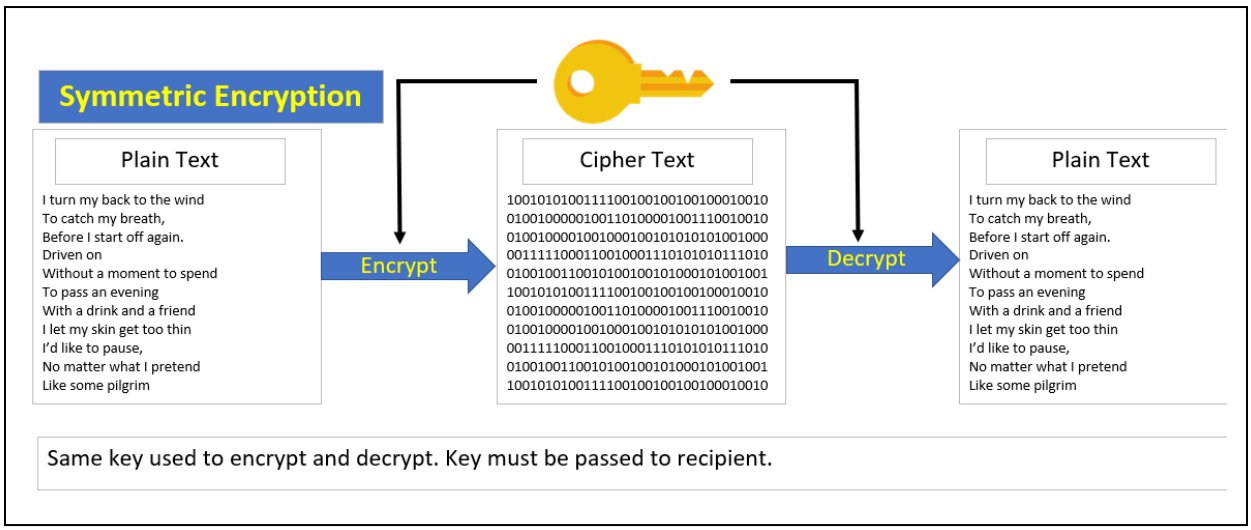
Section Content

Symmetric, Stream & Block Cipher Definitions

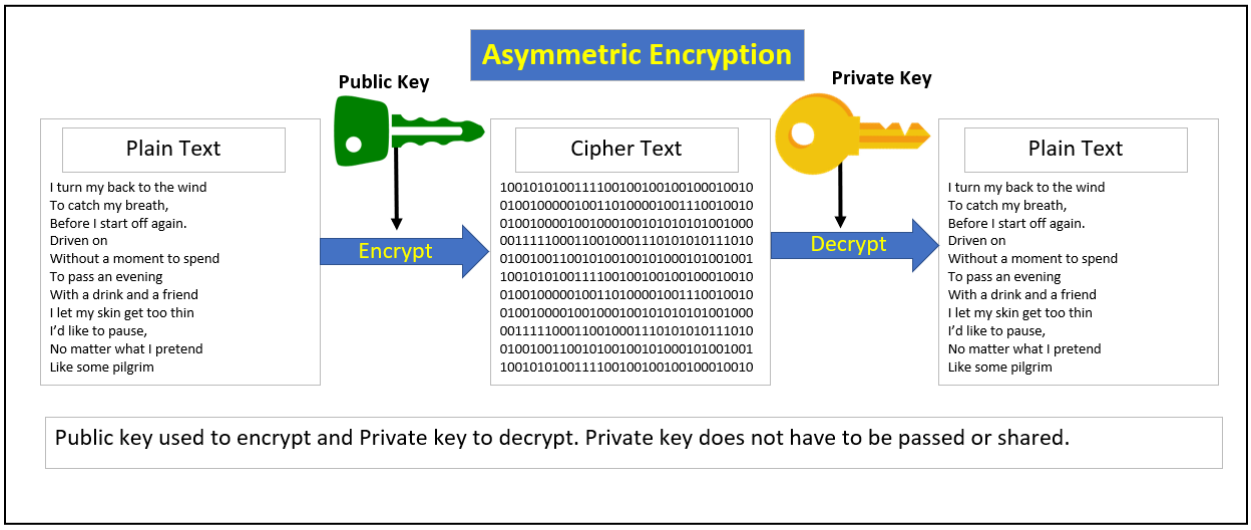
In this section you will learn about some terminology used for modern ciphers, and then learn the working details behind two ciphers. The first terms to learn about are symmetric and asymmetric ciphers.



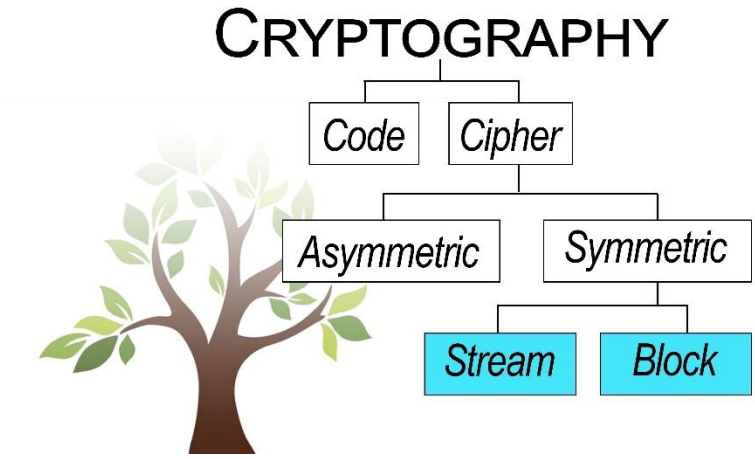
This is actually a technical way to say whether the cipher uses the same key to both encrypt and decrypt the message, in which case it's a symmetric key cipher, or if one key is used for encryption and a different key is used for decryption, in which case it's an asymmetric key cipher. Everything we've learned about to this point has been symmetric as all the ciphers from the Caesar cipher to the Vigenère and Vernam ciphers need the same key to decrypt the message that was used to encrypt it.



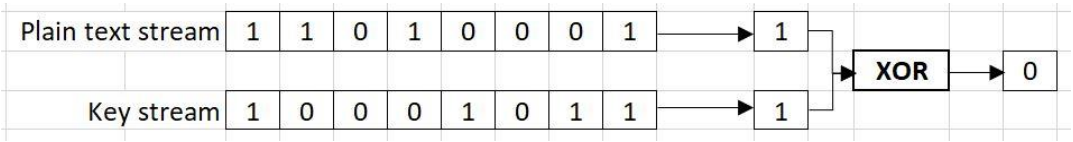
Asymmetric is a method of encryption that eliminates the need to exchange keys, and it's a critical component in providing secure communication on the Internet. That is, one key is sued to encrypt the data, and a completely different key is used to decrypt the data. With everything you've learned about encryption so far this may seem impossible, but some really smart people figured out a way to make something that seems impossible work. You will learn all about asymmetric encryption later in the class, but for now you just need to know what the term means.



Symmetric encryption is further sub-divided into two classes of ciphers, block ciphers and stream ciphers. These classes of ciphers are named for how much text they encrypt or decrypt at a time.



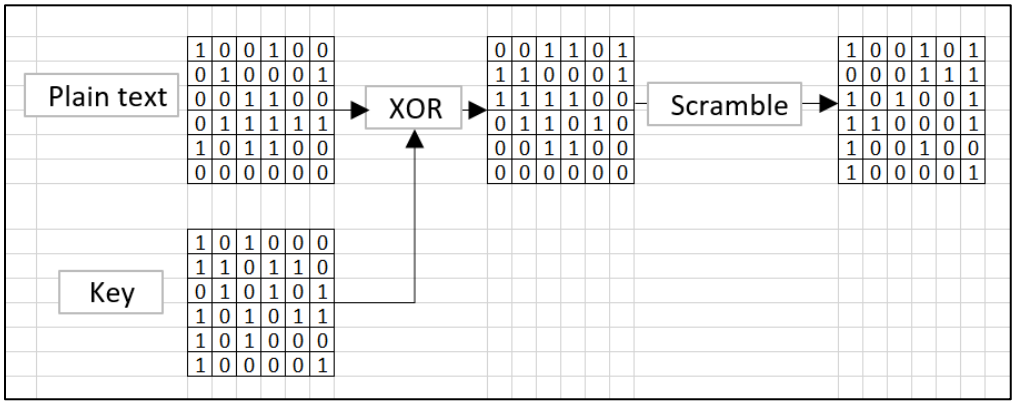
Stream ciphers get their name because they take all the data to be processed and queue it up in one long line. The data then flows, like water in a stream, one character or one bit at time into the encryption or decryption algorithm. When the encryption/decryption process is finished with one character, it takes the next character from the stream.



Stream ciphers process one bit or one character at a time.

All the ciphers you've learned about so far are stream ciphers. The substitution ciphers, transposition ciphers, and the Vigenère cipher all work on one character at a time; and the Vernam cipher works on one bit at a time.

With modern cryptography another class of ciphers called block ciphers was developed. The basic difference between a block cipher and a stream cipher is how many characters they process at a time. A block cipher doesn't process a single character at a time, instead it divides the plain text into blocks containing multiple characters and encrypts or decrypts all the characters in the entire block. The characters in the block will be scrambled around with other characters in the same block before XORing them. Once one block has been scrambled, the next block of data will be processed. In a way this is similar to a stream cipher, but once again block ciphers work on blocks of data instead of a single character or bit. You'll learn all the details about block ciphers in the next section, for now you just need to know the difference between stream ciphers and block ciphers, and why we call them that.



Block ciphers process many bits or characters at a time.

In the rest of this chapter you will learn the details about two stream ciphers, RC4 and Salsa20/ChaCha20. Both ciphers are basically Vernam ciphers that XOR the plain text with a set of key bits. The thing that makes each of these ciphers distinct is the PRNG they use.

As you learn about these ciphers, you'll see that it's possible to learn how they work on a basic level fairly quickly. At this basic level you'll learn the names of the systems and get a beginner's explanation of how each cipher works. I call this the management level of knowledge because in my experience most managers may know what something's called but won't have any clue, or any interest as to how it functions on a technical level. They also won't have any understanding of what the implications of the finer details may be. On the other end of the spectrum you can try

and learn everything about each of the ciphers and spend hours or days going deep into the math and the code used to implement the cipher. This will be helpful if you want to advance the science of cryptology but may not be the best use of your time if you don't want to specialize in cryptology.

I'm going to present three levels of details for RC4 and Salsa20/ChaCha20. The first explanation will be the simplest at the management level, the second level will provide a deeper explanation of the PRNG algorithms without going too deeply into the math, and the third level will be the full on geeked out math and code level explanation. I think that the second level will be a good level to understand if you are going to work in cyber security but don't plan on specializing in cryptography. In any case, I suggest that you concentrate on the general principles of both the RC4 and Salsa20/ChaCha20 PRNGs and try to not get too lost in details.

RC4 Stream Cipher

RC4 Implementation and History

RC4 was developed by Ron Rivest and stands for Rivest Cipher 4 or Ron's Code 4. Mr. Rivest, a well-known and highly respected cryptographer, originally developed RC4 1987 as a proprietary system to be used in conjunction with RSA public key encryption system which he also helped invent. The RSA system was one of the original components for secure communication on the web. RC4 was a key component of the RSA Public Key Infrastructure system for many years, but it's since been replaced. You'll learn the details about RSA and public key encryption later but suffice it to say that Ron Rivest knows what he's doing and RC4 is a widely respected cipher.

The code for RC4 is still considered a trade secret, and RSA had to sign an agreement with the NSA to keep it secret in order for the NSA to allow it to be used at all. However, in 1994 the source code was leaked by a group called the Cypherpunks and soon spread around the Internet. RC4 became very popular because it was much faster and efficient than the alternatives at the time, and it was very simple to implement in programming code. Some developers weren't 100% positive that the leaked code was the real RC4 algorithms, so they called their implementation Alleged RC4 or ARC4, which you may see on Linux systems. So even though RC4 was, and still is the intellectual property of RSA it soon became an integral part of many different applications and systems including:

- Secure Sockets Layer and Transport Layer Security (SSL/TLS) which are key parts of Internet security including HTTPS. RC4 was used with SSL/TLS until 2015.

- Wired Equivalent Protocol (WEP) or IEEE 802.11 which was the main security protocol for wireless routers and wireless networks for many years

- IBM used RC4 in its Lotus Notes / IBM Domino product, which is an email and collaboration platform, and desktop workflow application.

- Oracle still uses RC4 as an option

RC4 uses a Vernam cipher to do the actual encryption, so it's feature of interest is it's PRNG. Here an explanation of the RC4 PRNG, with three levels of detail.

Simplest Explanation of RC4 Algorithms

The RC4 cipher uses a variable key size, where the key must be at least 5 non-negative integers between 0 and 255 but can be as large as 256 integers. Most of the time we count the key size in bits. Each integer in the range 0-255 can be represented as an 8-bit number. That is, 0_{10} is the $0000\ 0000_2$ and 255_{10} is $1111\ 1111_2$ which is the largest number that can be represented by with 8 bits. This means that the RC4 key size, measured in bits would be 40-2048.

Once the key is entered the RC4 cipher works in three stages. The first part of the RC4 process is called the Key Scheduling Algorithm (KSA) where it takes the numbers 0-255, and then scrambles them based on a formula that uses the supplied encryption key to control the scrambling. You can think of this as shuffling the numbers like a deck of cards, or more like shuffling 5 decks of cards since there are 256 numbers instead of 52.In any case, the KSA puts the numbers 0-255 in an array and then scrambles them around using the RC4 Key Scheduling Algorithm.

The second part of the cipher is the PRNG portion. The PRNG uses the list of the scrambled numbers that was created during the KSA step. It goes to the first number in the list and swaps it with another number from the list, then returns this number as the first pseudo random number. If another pseudo random number is needed, then the PRNG goes to the second item in the list and does the same two item swap before returning the new number. This continues until the end of the list is encountered, and then it begins again at the beginning of the list. Going back to the playing card analogy, this is like grabbing the first card off the top of the deck but swapping it with another card and looking at the swapped card instead of the card you first grabbed. Doing this swap as you move through the card deck has the effect of performing another shuffle. This way when the end of the deck is reached the cards will have been shuffled again, and you won't have to wait to do another complete shuffle before starting over.

The third part of the cipher is where the pseudo random number is XORed with the first character from the plain text to create the cipher text. As we pointed out earlier, this is just the XOR from the Vernam cipher, so there's nothing new or unique about this step.

The hardest part of RC4 to understand is why the PRNG swaps the number in the list before it returns it. To explain this, let's look at two examples, one where the plain text is 256 characters or shorter, and one where the plain text is a lot longer than 256 characters. In either case, the KSA portion of RC4 scrambles or shuffles in the array of 256 numbers before anything happens, essentially creating a list of random numbers.

If the plain text only has 256 characters or less the scrambled numbers from the array of 256 numbers could be used as the pseudo random numbers and XORed with the plain text without any problems. The pseudo random numbers would act just like a one time pad and XORing them with the plain text would provide perfect security. But if the message is longer than 256 characters and the pseudo random numbers are repeatedly used in the same order then there will be a repeating pattern, which makes the whole scheme easy to break. To prevent the same

list of numbers from being used more than once each number from the list is swapped with another number in the list just before it's used.

This might seem like it's too simple to be secure because it's only using pseudo numbers between 0-255. Especially when you compare it with algorithms like LCG which can generate pseudo random numbers in a much larger range. The "trick" is that these 256 numbers are constantly shifting places, so they should never come up in the same order.

It's interesting that it turns out there are some problems with RC4. They're not huge problems and it just took over 20 years to find them. The fact that it took so long to find the problems just illustrates a couple of points about PRNGs. The first is that even though the mechanics of the RC4 PRNG are relatively simple, it's difficult to see how patterns in the results might occur. The second is that creating a CSPRNG is very difficult and should be left to the professionals.

Deciphering RC4 Encrypted Messages

Deciphering messages encrypted with RC4 is done by using the exact same process as encrypting the message. Remember that any bits encrypted with a key and the XOR function can be decrypted by XORing the cipher text bits with the same key. This means that in order to decrypt the message, the only other piece of information the recipient needs is the key. The recipient doesn't need the entire set of bits used to encrypt the message, if they have the key they can use the RC4 to regenerate the bits in the one time pad.

Medium Depth Explanation of RC4 Algorithms

Here's another explanation of RC4 that goes into more depth and provides additional details on the PRNG. The RC4 cipher uses a two stage process for generating pseudo random numbers. The first stage is called the Key Scheduling Algorithm (KSA), and the second stage is where the PRNG goes to work.

RC4 Key Scheduling Algorithm

The KSA starts by simply building an identity array of 256 integers and populates it with the numbers 0 – 255. Let's call this array S[ ] for now. The numbers in the S[ ] array will be initially set to 0 - 255 in order. In other words:

S[0] = 0
S[1] = 1
…
S[255] = 255

But these numbers will be used as the pseudo random numbers, so a critical step is to mix the numbers up. Plus, if the numbers are used without any scrambling then any attacker will be able to spot the pattern very quickly.

Scrambling the numbers in the S[ ] array is where the key comes in. This is accomplished by building another array, let's call it K[ ] and populating it with the numbers from the key. Remember that the key must be at least 5 numbers, but it can be as large as 256 numbers. So, for example if the key is set to the numbers 23, 47, 31, 73 and 7 then we will have:

K[0] = 23
K[1] = 47
K[2] = 31
K[3] = 73
K[4] = 7

The K[ ] array also needs to be of size 256, so if the key is shorter than 256 characters the key numbers are repeatedly used until the array is full.

When the K[ ] array is complete the scrambling of the S[ ] array begins. This step is very similar to shuffling a deck of cards, where the order of the cards is changed, but the deck starts and ends with the same cards. That is, we don't take any cards out and no new cards are added, it's the same cards but in a different order. In this case the numbers in the S[ ] array are shuffled and end up in a different order, but when it's finished the numbers 0 - 255 are still somewhere in the array. The actual shuffling is done by starting with the first element of the S[ ] array, and doing some calculations to pick another S[ ] array element to swap with. One of the factors in this calculation is the value in the corresponding element of the K[ ] array, so the results will be different for each different key. These calculations are a combination of two additions and two modulo operations between elements in the S[ ] and K[ ] arrays, but they will end up selecting a number between 0 – 255. This number is used to select a second number from the S[ ] array. Once the second S[ ] array element is selected, it is swapped with the first S[ ] array element.

As an example, say that the shuffling is still being done for array element S[0], and the calculation picks the number 31. This will result in the following swap:

tempVar = S[0]
S[0] = S[31]
S[31] = tempVar

This swapping process is then repeated for array element S[1], S[2] … S[255], which means that every number will be swapped at least once. However, some of the numbers will probably be swapped multiple times. At the end of this step the S[ ] array will still hold the numbers 0 – 255, but they will now be in a pseudo random order instead of being in numeric order. It's a pseudo random order because this order can be created again if the same key or K[ ] array is used, but it will be a completely different order if a different key is used.

RSA PRNG

Once the Key Scheduling algorithm is finished the RC4 cipher starts generating pseudo random numbers using it's own PRNG algorithm. The RC4 PRNG uses the S[ ] array which now holds the numbers 0 – 255 in a scrambled order.

The PRNG starts with the first number in the S[ ] array, S[0], and swaps it with another number from the S[ ] array, then outputs this number as the first pseudo random number. Yes, even though the numbers in the S[ ] array were just scrambled, an additional swap is performed before outputting the pseudo random number. The reason for this extra swap will be explained below.

Every time another pseudo random number is needed the PRNG algorithm moves to the next item in the S[ ] array and does the same extra swap before outputting the new number. This continues until the end of the S[ ] array is encountered, at which point the PRNG algorithm moves back to the beginning of the S[ ] array again.

The hardest part of this to figure out is why the PRNG swaps the number in the S[ ] array before outputting it. To explain this, let's look at two examples, one where the plain text is 256 characters or shorter, and one where the plain text is a lot longer than 256 characters.

If the plain text only has 256 characters or less the scrambled numbers from S[ ] array could be used as the pseudo random numbers and XORed with the plain text without any problems. The pseudo random numbers would act just like a one time pad and decrypting the message would require knowledge of the numbers in the S[ ] (or the key used to generate it). But if the message is longer than 256 characters and the pseudo random numbers are repeatedly used in the same order. This will result in a repeating pattern, which makes the whole scheme easy to break. To prevent the same pseudo numbers from repeating in a pattern each number from the S[ ] array is swapped with another number in the S[ ] array just before it's used.

It might help to view the problem using the card deck analogy again. In the key scheduling stage of RC4 the deck gets one good shuffle to pseudo randomize the numbers. We could view all the cards in the deck once and then give the entire deck another shuffle, or we could swap each card in the deck before we look at it. In either case the cards will be shuffled before we start viewing the deck again, but there's an added benefit in doing the continuous swapping.

If we wait until all the cards are viewed to do the shuffle, then each card will only show up once. The order of the cards will be randomized, but each card will only be seen once. If we swap cards before viewing each card it makes it possible to view a single card more than once. In fact, it's possible that we could see the same card all 52 times. This is of course highly unlikely but theoretically possible, like flipping a coin and coming up heads 23 times in a row. In math terms shuffling at the end results in 52! possible combinations, while swapping cards before viewing results in 52^{52} combinations.

$52! = 8.1e+67$
$52^{52} = 1.7e+89$

These are both enormous numbers, but 52^{52} is obviously larger. And if the point of the PRNG is to produce as many different random number patterns as possible then doing the continuous swap is the better option.

RC4 Enciphering

The actual enciphering in RC4 is a vanilla XOR between the pseudo random number and the plain text. The pseudo random numbers are integers between 0 and 255, so they will simply be represented as an 8-bit number. If the message contains plain text it will be represented as ASCII, and this 8-bit ASCII number will be XORed with the integer from the PRNG. The message can contain any type of data as the plain text data is XORed 8 bits at a time with the 8 bits representing the integer returned by the PRNG.

Explanation of RC4 Algorithms with Code

If you want to know the nitty-gritty details of the RC4 cipher, here they are. You don't need to know this for my class, but I'm presenting it just in case you're curious and want to see the code.

The RC4 uses the following pseudo code for the Key Scheduling algorithm. First the S[ ] array is populated using the following code:

```
for i = 0 to 255 do
      S[i] = i
```

Next read in the encryption key values, which consist of 5 – 256 non-negative integers whose values are between 0 and 255. Also read in the `keyLength`, which is the number of bytes or number of integers supplied as the key. Assume that encryption key values are stored in the array E[ ].

If the `keyLength` is less than 256, then we need to make it 256 by repeating the numbers provided as the key. This can be done using the following pseudo code which builds an array named K[ ] which will hold the key values:

```
for i = 0 to 255 do
      K[i] = E[I MOD KeyLength]
```

The next step is to perform an initial shuffle of the values in the S[ ] array, using the values from the K[ ] array as a key to control the scrambling. This is accomplished with the following code:

```
for i = 0 to 255 do
      S[i] = i
      j = 0
      for i = 0 to 255 do
            j = (j + S[i] + K[i])(mod 256)
            swap (S[i], S[j])
```

At this point we can start to generate the pseudo random numbers, or keystream, from the values in the S[ ] array. The following code steps through each element of the S[ ] array, and swaps it with a different element before doing one more MOD and returning the result as the pseudo random number:

```
i, j = 0
while ( true )
      i = ( i + 1 ) mod 256
      j = ( j + S[i] ) mod 256
      SWAP S[i], S[j]
      k = ( S[i] + S[j] ) mod 256
      output S[k]
```

Note that the key is no longer involved. It was only used during the initial shuffle of the K[ ] array.

RC4 Implementation Problems

In actuality RC4 is just a PRNG, or at least the only novel part of the cipher is the PRNG, because it uses a Vernam cipher to do the actual encryption. And while the code for the PRNG is pretty simple, like any PRNG, it's difficult to look at the code and make the determination that

it's safe to use or discern if there are any inherent problems. As it turns out there were a few problems with RC4's PRNG, but it took over 20 years to identify them. Some of these problems aren't with RC4 so much as they are with the way that different groups or companies chose to implement the code. This is one of the key lessons in cryptology that was introduced in the beginning of the class. That is, parts of cryptology, like implementing cryptographic systems, can be extremely difficult and should be left to someone who really knows what they're doing.

One of the features of RC4 that's caused problems is that it has a variable key length, from 40 bits or 5 numbers up to 2048 bits or 256 numbers; and some implementations use the shortest key possible. Or some companies set up their encryption systems to use the same key repeatedly, which would cause problems with any implementation.

For example, consider the Wireless Encryption Protocol (WEP) which uses RC4. You might remember WEP if you're old enough as it was the original protocol used by wireless routers. But it turned that the WEP implementation was engineered to use a short key, only 40 bits, which is combined with a 24-bit IV to create the RC4 key. And even worse it uses the same key repeatedly as packets are sent between the wireless router and any connected device. With only 24 bits in the IV there is a greater than 50% chance it will be repeated after ~5000 network packets are sent. An attacker can easily send this many packets to a wireless router in a few minutes, and once the IV and the key are repeated gain access to the key. This makes wireless routers that use WEP extremely vulnerable; the router password can typically be cracked in a few minutes using easily obtainable software such as aircrack-ng[1]. Again, this isn't caused by a problem with RC4, it's caused by the poor choices someone made when implementing the RC4 algorithms. No encryption scheme will be secure if the keys are used repeatedly. The failure of WEP is a good illustration of the point that anyone implementing a cryptographic solution needs to have a strong understanding of the algorithms being used.

In any case, researchers have identified several attacks that would work against the RC4 cipher and it's now suggested that it not be used. There are counter measures that can be taken to strengthen RC4 implementations against most of these attacks, but the consensus seems to be that RC4 should not be used.

Salsa20 and ChaCha20

The next ciphers you will learn about are Salsa20 and its descendant ChaCha20. They were written by a Daniel Bernstein as part of eStream, which was an effort in the early 2000's sponsored by the European Union to find a stream cipher that everyone could use. (As you'll learn later, this is similar to what NIST did earlier in the United States.) At the time this was written, during the fall of 2020, ChaCha20 is the main replacement for RC4 and is widely used on the Internet.

The general design of both of the Salsa20 and ChaCha20 ciphers is the same, they both have a PRNG section that uses an 8 byte x 8 byte table to generate 512 pseudo random bits which are

[1] https://github.com/aircrack-ng/aircrack-ng

XORed with the plain text bits to perform the encryption. The things about these algorithms worth inspecting are the algorithms used in the PRNG section as they're different than anything you've seen previously. While the PRNGs for the two ciphers are very similar, there are some differences in the technical details. You'll learn the details of the ChaCha20 cipher as it's the most modern, and at the end you'll learn the differences in the Salsa implementation.

Explanation of the ChaCha20 PRNG

ChaCha20 starts by taking a 256 bit encryption key and writing it into the middle rows of a square matrix. The matrix is 64 bits x 64 bits, or 8 bytes x 8 bytes. While the key is 256 bits or 32 bytes, you can also think of it being 32 numbers that are between 0 – 255, so each number can be stored in a single byte. For example, if we used the numbers 1 – 32 as the key the matrix would look like this after the key was loaded. (Of course, this would be a horrible key, but using these numbers makes it easy to visualize how this table is being built and where the key bytes are placed.)

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	26	27	28	29	30	31	32

The first two rows of the table are filled with a 16 character constant which is always the string "**expand 32-byte k**". Remember that the binary ASCII values will actually be written to the table, but I'm going to write the text characters to once again make it easier to visualize.

e	x	p	a	n	d		3
2	-	b	y	t	e		k
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	26	27	28	29	30	31	32

The next section of the table is filled by the bytes for two 32 bit counters. These counters keep track of how many times the table has been used to generate the set of 4096 pseudo random bits. The figure shows where the bits for the first counter C1, and the second counter C2 are located in the matrix.

e	x	p	a	n	d		3
2	-	b	y	t	e		k
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	26	27	28	29	30	31	32
C1		C2					

The last 64 bits are used to hold something called a nonce. Most definitions of the word nonce say it means something that is only used once. Although in Britain, it's slang for sex offender, so be careful if you talk cryptography with anyone from Britain! The nonce is very similar to the key, with one slight but important difference. Both the nonce and the key are used to control the encryption process, in this case they're both used in the generation of the bits for the keystream. And the sender must transmit both the nonce and the key to the recipient so the message can be decrypted. So, in some sense both the nonce and the key are part of the encryption key. The difference between the nonce and the key is how they're generated. Generally speaking, the key is a set of random numbers which are grabbed from some true random number generator, while the application itself generates the nonce, typically by using a counter, so that it can ensure that it's not repeated. For example, the first time a program using ChaCha20 encrypts a message it sets the nonce to say 41. This nonce will be used to for the entire message. But when that same program is used to encrypt a second message it will increment the nonce to 42.

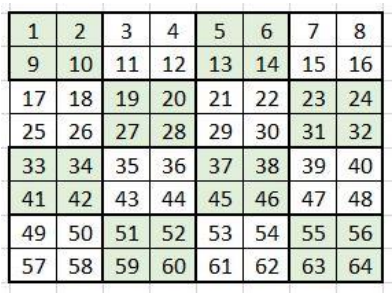
Using a nonce makes it prevents problems that would occur if you encrypt the messages using the same key. Remember that using the same key for multiple messages makes it easy to attack and decipher the messages. By introducing the nonce into the PRNG, the key bits will be changed even if the same key is provided.

e	x	p	a	n	d		3
2	-	b	y	t	e		k
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	26	27	28	29	30	31	32
				Nonce			

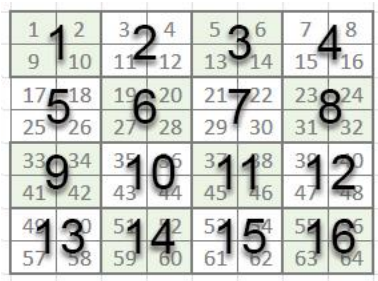
After the nonce is added the matrix is fully populated and ready for the next step, which is where the contents of the cells get scrambled and mixed. To help visualize the how the scrambling works I'm going to place numbers in each cell of the matrix. Remember the matrix cells will actually hold 8-bit chunks of binary data that represent ASCII characters or numbers, not these numbers. But using the numbers should really help you see how the scrambling works.

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	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	52	53	54	55	56
57	58	59	60	61	62	63	64

The first scrambling step is to divide the matrix into sixteen sections that will each hold 4 cells of the matrix. Since each cell contains 8 bits of binary data, the entire 4 cell section will hold 32 bits of binary data. The 16 sections are shown by the colored blocks in the following figure.



The next step in the scrambling is to mix and match the data in the cells into blocks. ChaCha20 does this by combining the data in 4 cells, which results in a new smaller 4x4 matrix. In the new matrix each block will hold 32 bits of data since this is a combination of 8 bits of data from each of the 4 original cells. The data in block 1 of the new matrix will be the bits from cells 1, 2, 9 and 10 of the 8x8 matrix; the data in block 2 of the new matrix will be the bits from cells 3, 4, 11 and 12 of the 8x8 matrix, etc. Using the same numbering scheme results in a matrix that looks like the following, where the original cell numbers are in the small font and the block numbers are in the large font. Remember that the numbers aren't what's being stored in the cells, they're just labels to help us keep things straight.



At this point some repetitive rounds of scrambling are performed by sub-dividing each chunk of the 4x4 matrix into groups of 4 blocks, which contains 16 of the original cells. These groups are called quarters because the block is a quarter of the original matrix. For example, quarter 1 will contain the blocks 1, 2, 5 and 6, or the cells shown below.

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

ChaCha20 performs10 rounds of this quartering and scrambling by doing the following:

A. First the chunk of the matrix is further sub-divided into 4 columns. In the case of the first 4x4 chunk the columns will look like this:

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

B. The numbers in each column are then changed by adding some of the numbers together, XORing other pairs, and shifting the bits in the binary number in some of the numbers. The easiest way to illustrate this is to assign a letter to each cell in the columns. In this case cell 1 becomes **a**, cell 9 is mapped to the letter **b**, cell 17 is mapped to **c**, and cell 25 is assigned the letter **d**.

a
b
c
d

The scrambling of the numbers in each column is done with these 12 operations:

```
a = a+b
d = d XOR a
d <<<= 16      (This shifts the bits in the number d 16 places to the left)
c = c+d
b = b XOR c
b <<<= 12
a = a+b
d = d XOR a
d <<<=  8
c = c+d
b = b XOR c
b <<<=  7
```

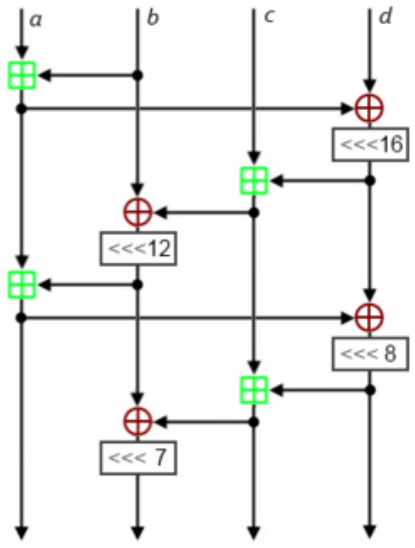
The <<<= is the bit shift operator, which in this case says to shift or rotate bits to the left. When this happens, all the bits in the number are shifted the specified number of places. If the operator uses <<< then the bits are shifted to the left, if it uses >>> then they are shifted to the right. Bits that get pushed off the end of the number will be wrapped back around to the beginning of the number.

Take for example 11000011_2 `<<<= 2` This says to shift the bits two places to the left. Instead of just shoving the left two most bits off the left end, they are wrapped back around and placed on the right, resulting in: 00001111_2

The C/C++ family of programming languages use what they call shorthand operators for the addition and XOR operations. The addition shorthand is `+=`, so `a+= b` is the same as `a = a+b`. The shorthand for the XOR operation is `^=`, so `d ^= a` is the same as `d = d XOR a`. Using the shorthand operators and writing three operations per line makes it a little easier to look at the operations and see what's happening:

```
a += b;  d ^= a;  d <<<= 16;
c += d;  b ^= c;  b <<<= 12;
a += b;  d ^= a;  d <<<=  8;
c += d;  b ^= c;  b <<<=  7;
```

Or it may also help to look at this picture from Wikipedia to see how these operations are chained together. First **a** is modified, then **d**, then **c**, followed by **b**; and the process repeats until all the cells are modified twice.



It's important to note that this scrambling happens to the numbers in all four columns. Each time a new column is scrambled the top cell is assigned to **a**, the second cell down is assigned to **b**, the third cell down to **c**, and the bottom cell is assigned to **d**.

C. After the column scrambling is finished the 16 number matrix is sub-divided again, but this time into 4 diagonal sections instead of into columns. If you just look at the algorithm for doing this diagonal sub-division it can be confusing, but when you see a picture of how it's done it's pretty straightforward.

Here's the first diagonal. It consists of cells 1, 10, 19, and 28.

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

The second diagonal has to wrap around to get the 4th cell. It consists of cells 2, 11, 20 and 25. The diagonal might be easier to see if we place a duplicate of the table next to the original table:

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

				duplicate			
1	2	3	4	1	2	3	4
9	10	11	12	9	10	11	12
17	18	19	20	17	18	19	20
25	26	27	28	25	26	27	28

The third diagonal also has to wrap around to get the last two numbers. It consists of the cells 3, 12, 17 and 26. These figures show the single table, as well as the duplicated table that makes it easier to see the diagonal.

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

				duplicate			
1	2	3	4	1	2	3	4
9	10	11	12	9	10	11	12
17	18	19	20	17	18	19	20
25	26	27	28	25	26	27	28

The fourth diagonal must also wrap around to get the last three numbers. It consists of the cells 4, 9, 18 and 27. These figures show the single table, as well as the duplicated table that makes it easier to see the diagonal.

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

				duplicate			
1	2	3	4	1	2	3	4
9	10	11	12	9	10	11	12
17	18	19	20	17	18	19	20
25	26	27	28	25	26	27	28

D. After the four diagonals are set, the four numbers in the diagonals are scrambled using the same 12 formulas used to scramble the numbers in the columns. This scrambling happens to the numbers in all 4 diagonals.

E. The complete set of 12 scrambling operations is called a round. At this point one round of scrambling has been done on each of the 4 column groups, and another round has been done on each of the 4 diagonal groups. The entire scrambling procedure is done nine more times, which means there will be a total of 10 rounds of column group scrambling and 10 rounds of diagonal group scrambling, for a grand total of 20 rounds. This is where the 20 in the Salsa20 and ChaCha20 comes from.

After the 20 rounds of scrambling the values in the matrix should be completely mixed up. However, since all the functions used to scramble the values can be reversed or inverted, it's possible to work backwards from this scrambled matrix to rebuild the original matrix. To prevent the entire process from being invertible one final step is performed where the values from the original unscrambled matrix cells are added to the values in the scrambled matrix cells. This way, the only way to work backwards would be if you knew the original matrix values, which means knowing the original key and the nonce.

When all the scrambling is finished the matrix holds 512 bits, called a keystream, which can be used to XOR with the plain text.

Whenever more keystream bits are needed the scrambling process is repeated with one minor variation. The 8 byte x 8 byte matrix is reloaded with the constant string, reloaded with the same key bits, and reloaded with the same nonce; but the counter is incremented by one before it's loaded into the matrix. This means that the only thing that will be different at the start of the scrambling is 1 bit in the counter section. This doesn't seem like it would be enough to make a big difference between one set of keystream bits and the next, but apparently it's enough to make a huge difference.

Differences between Key, Nonce and Counter

One of the things that I originally found terribly confusing is the difference between the key, the nonce and the counter. They're all used to control the encryption process so I couldn't see why they weren't just combined to make the key bigger. It turns out that they need to be separate as they all perform a different function. Here's an explanation of each including how it's generated and how it's used.

Key – I'm pretty sure you're familiar with the key, but let's list the characteristics so they can be used in comparison with the counter and the nonce. The key is generated by obtaining a true random number, usually from something like that measures true entropy like an outside source that measures radioactive decay. The key, the same key, is used throughout the encryption process for an entire message. But when a new message is encrypted a new random key must be generated for use with each new message. The last characteristic is that the recipient needs to know the key to decrypt the message, so the key must be transmitted along with the message.

Nonce – The nonce is similar to the key in that a single nonce is used to encrypt an entire message, and each new message requires the generation and use of a new nonce. The recipient also needs to know the nonce to decrypt a message. However, unlike the key, which is a random number, the nonce is not random. Instead, the nonce is generated by the encryption program which tracks the numbers it uses each time it runs. The application uses a counter to generate the nonce, so that it can guarantee that the same nonce is not used. It may seem like the nonce will be repeated sooner or later and create a pattern, which would be a problem. But while it's true that the nonce will repeat there are 64 bits to hold the nonce which means that the nonce numbers can range between 0 and 2^{64}-1, which works out to over 1.8e19 values. If you encrypted 1 message per second it would take over 584 billion years before the nonce repeated.

I'm not exactly sure who came up with the idea of using a nonce, or why it's used at all. But my guess is that it's used as a safety measure for implementation problems that vexed RC4. That

is, RC4 was a good cipher, but if you implement it with short keys and repeat the use of the keys then of course it's going to be weakened. Adding a nonce to Salsa20/ChaCha20 allowed Mr. Bernstein to ensure that even if a message was encrypted twice with the same key the resulting cipher text would be different. It's a guarantee that even if you repeatedly use the same key, the key plus the nonce will be different.

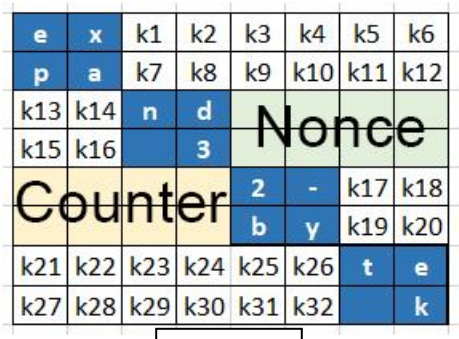
Counter – The counter is also generated by the encryption program and like its name implies it's a counter not a random number. The difference between the counter and the nonce is that the counter is incremented each time a new matrix of key bits is generated, so it may change hundreds or thousands of times for each single message. Compare this to the nonce which will remain unchanged in each single message. When a new message is encrypted the counter will be reset to 0, while the encryption program will figure out what value the nonce was set to for the previous message, increment it by one, and use the incremented value as the nonce while encrypting the entire new message. The other characteristic of the counter is that the recipient does not need to know the counter value. This is because the counter will always be set to 0 at the start of decryption, and then incremented each time a new matrix of key bits is required.

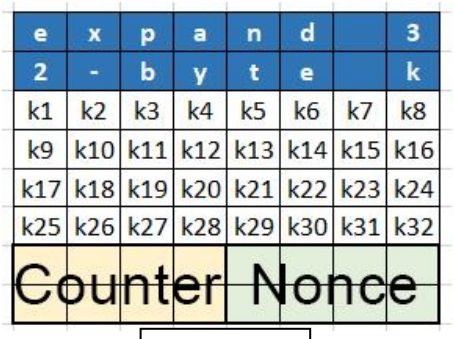
	Key	Nonce	Counter
Random or Sequential	Random	Sequential	Sequential
Frequency of Change	Once per message	Once per message	Every 512 bits
Required by Recipient	Yes	Yes	No

Differences Between Salsa20 and ChaCha20

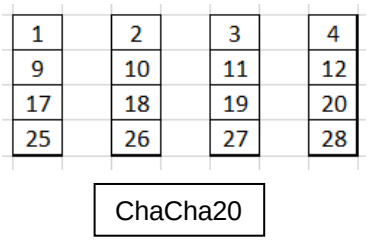
Mr. Bernstein wrote the Salsa20 cipher first, and after several years of working with it he found that the basic design was sound, but that it would run faster with a few tweaks. The result of these changes is the ChaCha20 cipher.

The first difference between the ciphers is how the initial matrix is loaded with the constant string, the key, the counter and the nonce. Each piece of data is still the same number of bytes as in ChaCha20, but in Salsa20 they are loaded in different cells in the matrix as shown in the figure. The same constant "**expand 32-byte k**" is used, but it's written diagonally across the middle of the matrix. The key numbers k1 – k 32 are split and written in some cells in each of the rows. The same size nonce is used, but it's written to the cells in third and fourth rows. The same size counter is used, but it's written to cells in the fifth and sixth rows.





The only other significant difference is that during the rounds salsa first divides the matrix into rows instead of columns.



Summary

You've been presented with a lot of details about the RC4 and Salsa20/ChaCha20 ciphers, and I realize it can be a little confusing. To help you out here are the general concepts that I believe are the important things to take away from this chapter.

1. There are two main classes of modern ciphers, symmetric and asymmetric. Symmetric ciphers use the same key to both encrypt and decrypt a message.

2. There are two classes of symmetric ciphers, stream and block. Stream ciphers encrypt 1 bit at a time, while block ciphers encrypt a block of bits.

3. Both RC4 and Salsa20/ChaCha20 are symmetric ciphers, which means the same key is used to encrypt and decrypt the message. They are also both stream ciphers, which means they work on a stream of bits, one bit at a time.

4. Both RC4 and Salsa20/ChaCha20 are Vernam ciphers, which means they XOR the plain text bits with the key bits. The unique feature of both ciphers is how they generate the key bits, or what their PRNG algorithms look like.

5. The RC4 PRNG takes a key from 40 bits – 2048 bits (5 integers to 256 integers in the range 0-255). RC4 uses the key to control the shuffling of 256 integers, from 0-255. The shuffled numbers are used as the key stream. The key numbers are continuously shuffled by swapping two of them each time a new number is used.

6. The recipient of a message encrypted with RC4 needs the key to decrypt it.

7. RC4 has some technical issues that make it vulnerable, so it is no longer used. However, the biggest issue with RC4 implementations were bad choices made by those doing the implementation. For example, WEP implements RC4 with short keys, and repeats the use of those keys making it easy to break.

8. Salsa20/ChaCha20 requires a 256-bit key, and a 64-bit nonce. This information is placed in a matrix, along with a constant string and a counter. The 512 bits in the matrix are scrambled 20 times and then used as a keystream to XOR with the plain text bits.

When more bits are required the matrix is reloaded with the key, nonce and constant, and the incremented counter; and the 20 rounds of scrambling are performed again.

9. The recipient of a message encrypted with Salsa20/ChaCha20 requires the key and the nonce to decrypt it.

10. A nonce is like a key, but instead of being completely random it is usually a sequential number generated and tracked by the application.

11. For many years RC4 was used in SSL/TLS, which are at the heart of Internet encryption. It has been replaced by ChaCha20.