5 Symmetric Cryptography: Stream Ciphers

Introduction

In the previous section you learned about the main building blocks of a form of modern encryption, the XOR function and CSPRNGs. These building blocks make it possible to achieve the same perfect protection achieved with the Vigenère cipher, but on any type of digital data. In this section you will learn about modern encryption ciphers that use their own unique CSPRNG code to generate the necessary key bits. The encryption algorithms you'll learn about in this chapter are in a class of ciphers known as symmetric stream ciphers. In this section you will first learn the definition of symmetric encryption, and about the two main forms of symmetric encryption which are stream ciphers and block ciphers. Most of the chapter is dedicated to teaching you about the specifics of the CSPRNGs used by two stream ciphers, RC4 and Salsa20/ChaCha20.

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 can weaken its 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 Salsa20and chacha
https://ianix.com/pub/chacha-deployment.html - software and systems using ChaCha

Section Content

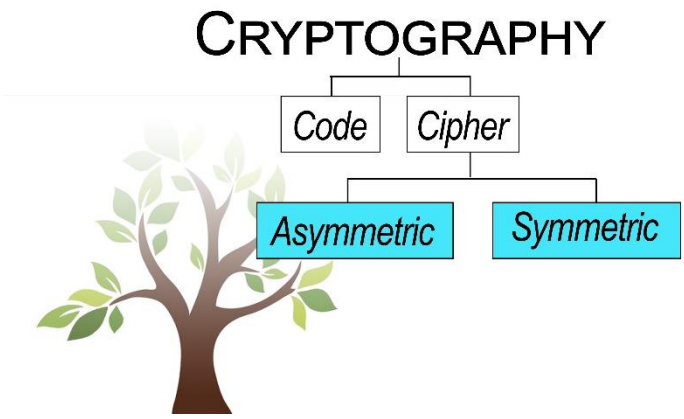
Introduction

Let's start this chapter with a quick recap of what you've learned about encryption to this point. The first thing you learned was how the first encryption was done using substitution ciphers which are easy to crack using frequency analysis. This led to the evolution of poly alphabetic ciphers such as the Trithemius cipher and the Vigenère cipher. The Vigenère cipher requires a key to encrypt and decrypt messages, but it can also provide perfect protection if the keys meet certain requirements. The main problem with the Vigenère cipher is that it only works with text, so it can't be used for any other type of data. The next step in the evolution of cryptography was the Vernam cipher which works on any binary data. The Vernam cipher replaces the tabula recta with the XOR function. The Vernam cipher also uses a key, and can also provide perfect protection if the key meets specific requirements. The name for the algorithms used to generate the key bits is Cryptographically Secure Pseudo Random Number Generator or CSPRNG.

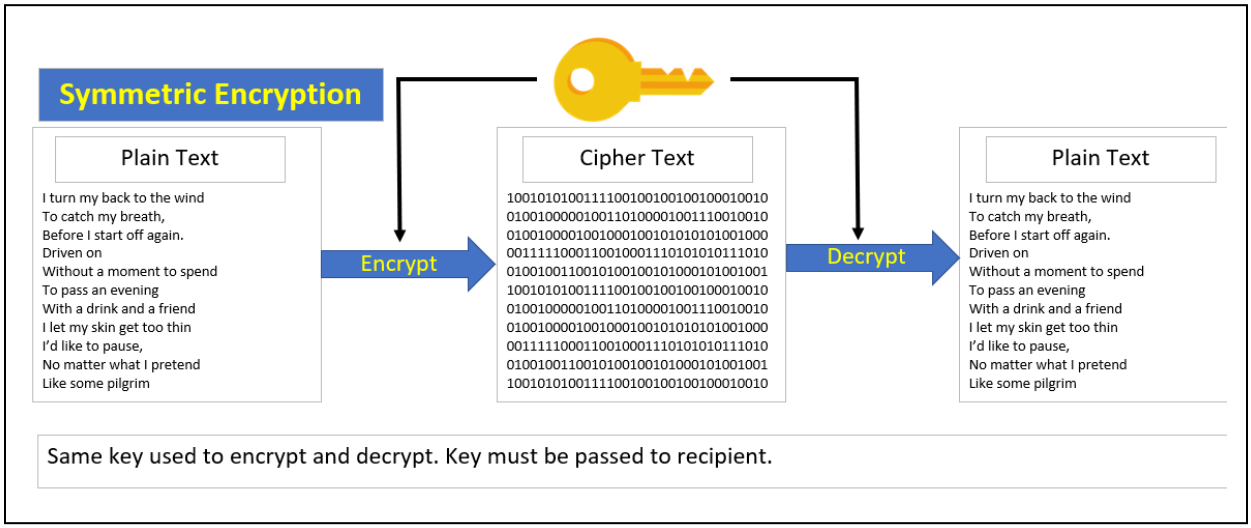
In this chapter you will learn about a set of encryption algorithms that use the XOR function and their own CSPRNG code. The names of these encryption algorithms are RC4 and Salsa20/ChaCha20. Before you learn about these algorithms you'll learn some additional terminology that's used to categorize algorithms that use the Vernam cipher, the XOR function and a CSPRNG.

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.

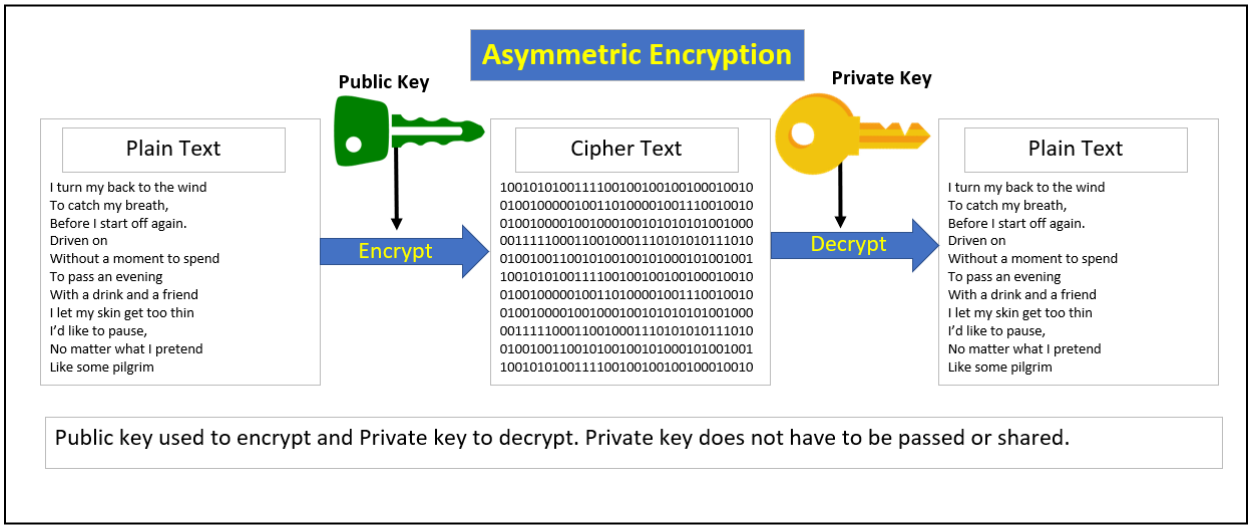


This is actually a technical way to say whether the cipher uses the same key to both encrypt and decrypt the message, in which case it's a symmetric key cipher, or if one key is used for encryption and a different key is used for decryption, in which case it's an asymmetric key cipher. This might seem like a strange distinction as everything we've learned about to this point have been symmetric ciphers. 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. However, all symmetric ciphers share the same problem which is finding a way to securely exchange the key.

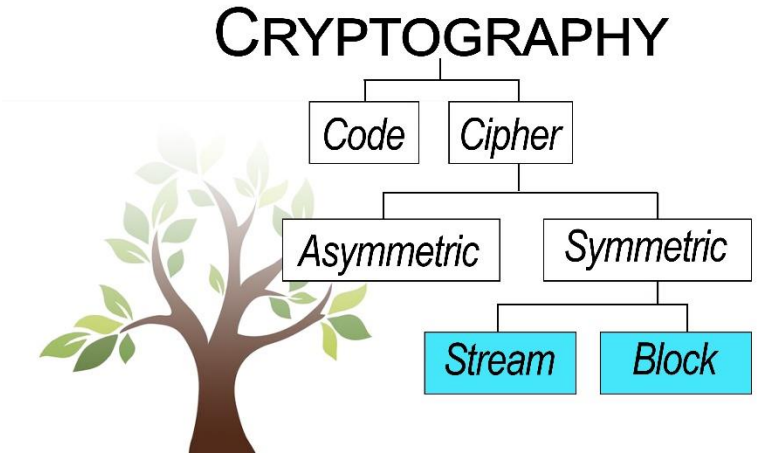


The key exchange problem vexed cryptographers for centuries, but it was finally solved using something called asymmetric encryption. Asymmetric encryption is a method of encryption that one key is used to encrypt the data, and a completely different key is used to decrypt the data.

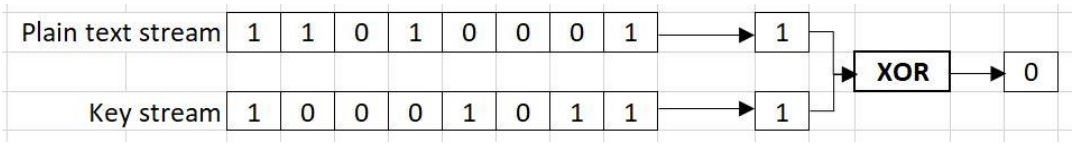
Using different keys eliminates the need to exchange keys, which has made asymmetric encryption a critical component in providing secure communication on the Internet. With everything you've learned about encryption so far using different keys to encrypt and decrypt may seem impossible, but some very intelligent 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.



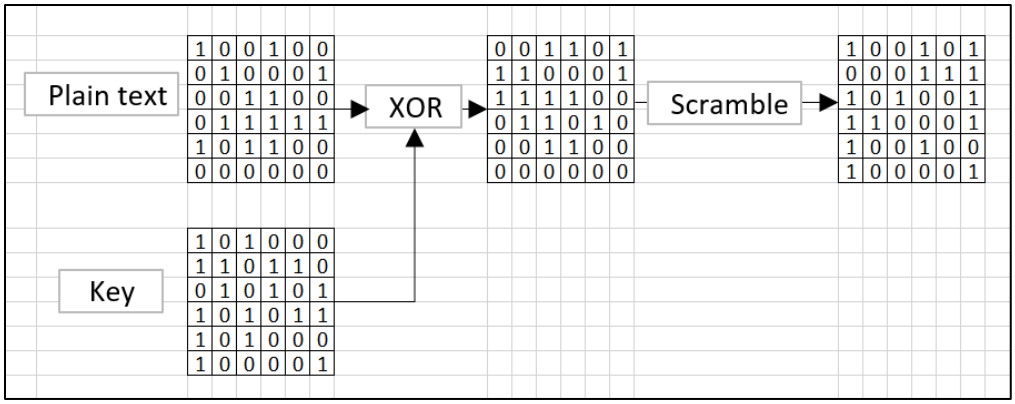
Stream ciphers get their name because they take all the data to be processed and queue it up in one long line. The data then flows, like water in a stream, one character or one bit at time into the encryption or decryption algorithm. When the encryption or decryption process is finished with one character, 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 about how it functions on a technical level. They also won't have any understanding of what the implications of the finer details may be. This level is also good for gaining a basic understanding of how things work before jumping into and possibly getting lost in the technical details. 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. My personal opinion, for what it's worth, is 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

The first stream cipher we'll look at is called RC4. 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, which means it works by using a CSPRNG to generate random bits for the key, XORs the key bits with the plain text to encrypt a message, and XORs the key bits with the cipher text to decrypt the message. This means looking at RC4's technical design from a big picture perspective won't show anything new, it just uses a CSPRNG and the XOR function. The feature of interest, the one we want to look at closely is it's CSPRNG. Here an explanation of the RC4 CSPRNG, with three levels of detail.

Simplest Explanation of RC4 Algorithms

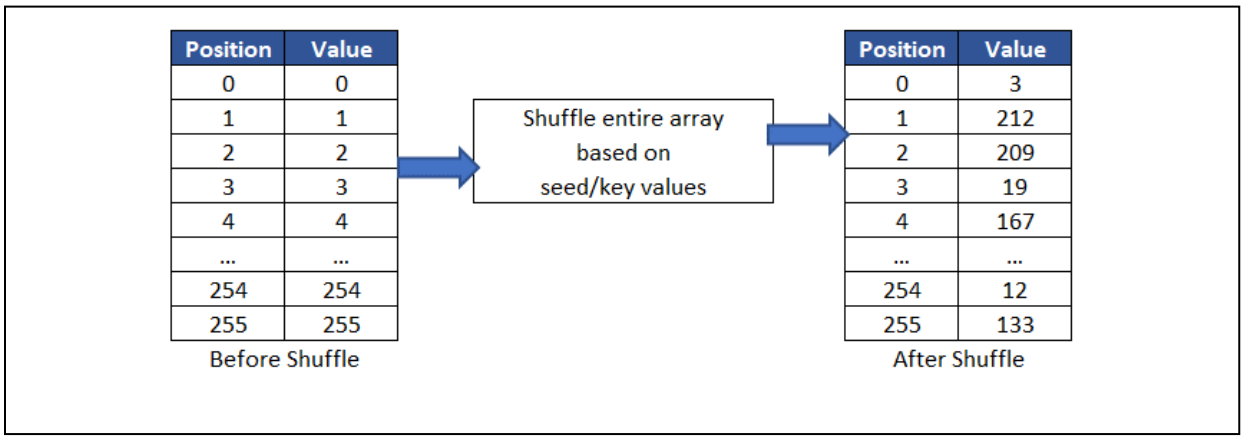
This section will provide the simplest explanation of RC4. It will be light on technical details but should provide an overview of how the RC4 PRNG functions.

The first thing to talk about is the size of the seed used for the RC4 PRNG. However, at this point we're going to change our terminology and call the seed the key. I know this is confusing as this means we're using the term key to mean two different things. Up to this point we've been using the term key to refer to the set of characters or bits that are used to encrypt the plain text or decrypt the cipher text. We've also used the term seed to refer to the set of numbers passed to a PRNG to determine or control the set of bits that will be generated. The seed is also the information that must be passed between the sender and the recipient to allow the recipient to get the PRNG to generate the set of bits necessary to decrypt a message. But like I said cryptographers also refer to the seed as the key. This mainly call it the key when they talk about the information that must be passed from the message sender to the recipient. In any case, referring to the seed as the key can be confusing, but you can look at the context you should be able to figure out which meaning should be used.

Ok, back to the RC4 cipher. The RC4 cipher allows for a variable size seed for it's PRNG. The seed/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

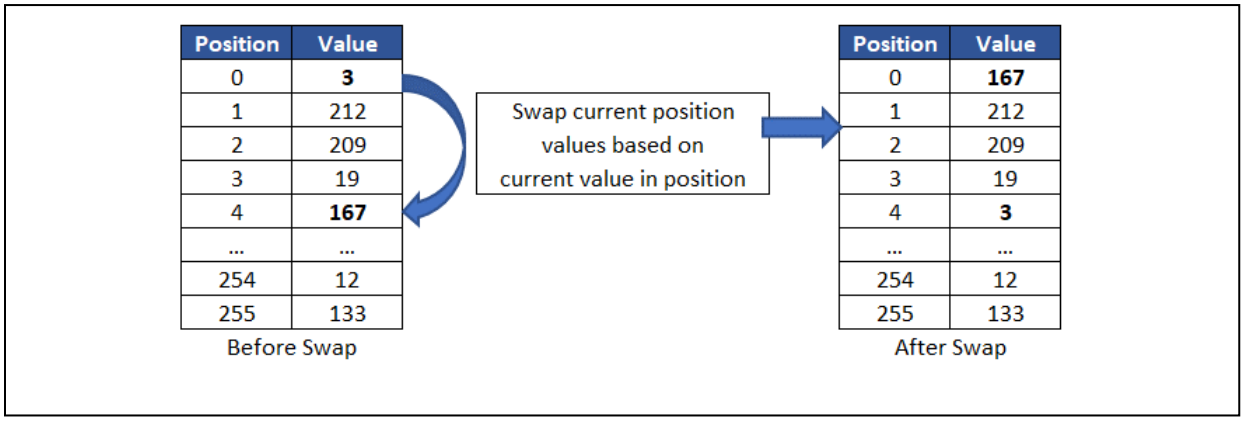
be represented as an 8-bit number. That is, 0_{10} is the 0000 0000_2 and 255_{10} is 1111 1111_2 which is the largest number that can be represented by with 8 bits. This means that the RC4 key size, measured in bits can be as small as 5 numbers or 40 bits, or as large as 256 numbers or 2048 bits.

Once the key is entered the RC4 PRNG works in three stages. The first part of the RC4 process is called the Key Scheduling Algorithm (KSA) which takes the integer numbers 0-255, and then scrambles them based on a formula that uses the supplied seed/key numbers to control the scrambling. That is, RC4 is always going to use the numbers 0-255, but it uses the seed/key numbers you supply to determine exactly how to scramble them. This is like shuffling the numbers 0-255 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 256 integers 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. The position of the second number in the swap is determined by the value stored in this position. For example, if the first number in the seed/key array is 27 then 27 will be plugged into the calculation to determine the second position. For this example let's assume the result of the calculation is 28. This means the numbers in positions 0 and 28 are swapped. After this swap the PRNG returns the number in position 0 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 using the number stored in the

second position to determine the location of the second number to swap, finally returning the number in the second position.



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 overall effect of performing another shuffle while you look at the cards. 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 one thing that should be made clear is that we've been talking about the RC4 PRNG working with the integers 0-255 and returning individual numbers, but when the XOR function deals with bits, not integer numbers. This is accomplished by treating each number returned by the PRNG as an unsigned integer and just using the 8 bits that represent the integer number. For example, 0_{10} = $0000\ 0000_2$ and 255_{10} = $1111\ 1111_2$.

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 uses the seed/key to scramble or shuffle the array of 256 numbers once 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.

That's the simple explanation of the RC4 PRNG. It's much different than the LCG PRNG in that it scrambles the same numbers instead of always calculating new numbers. This might seem like it's too simple to be secure because it's only using 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 point of emphasis 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 algorithm to regenerate the numbers needed to decrypt the message.

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 seed/key. Remember that the key must be at least 5 numbers, but it can be as large as 256 numbers. For example if the key is set to the numbers 2, 47, 31, 254 and 7 then we will have:

```
K[0] = 2
K[1] = 47
K[2] = 31
K[3] = 254
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.

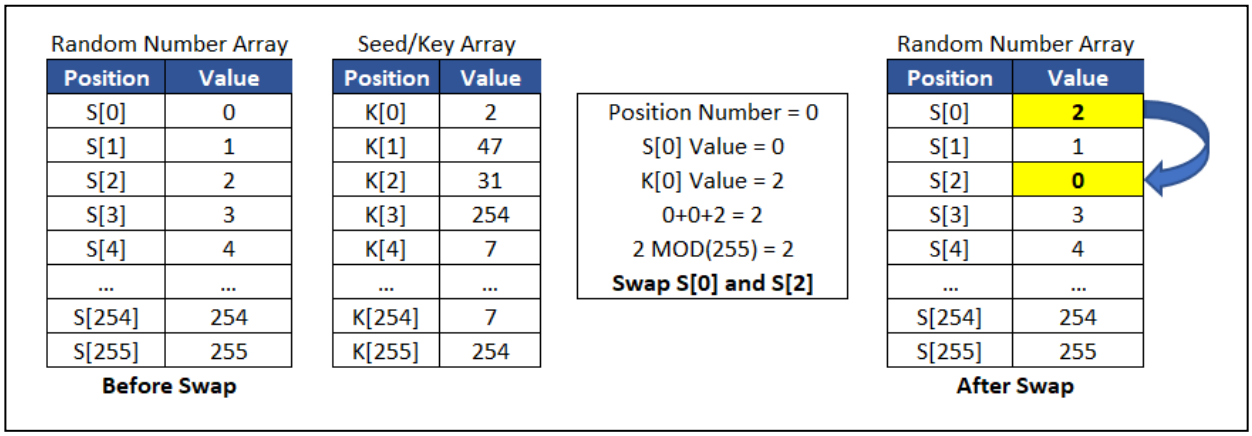
```
K[0] = 2
K[1] = 47
```

```
K[2] = 31
K[3] = 254
K[4] = 7
K[5] = 2
K[6] = 47
K[7] = 31
K[8] = 254
K[9] = 7
…
K[254] = 7
K[255] = 2
```

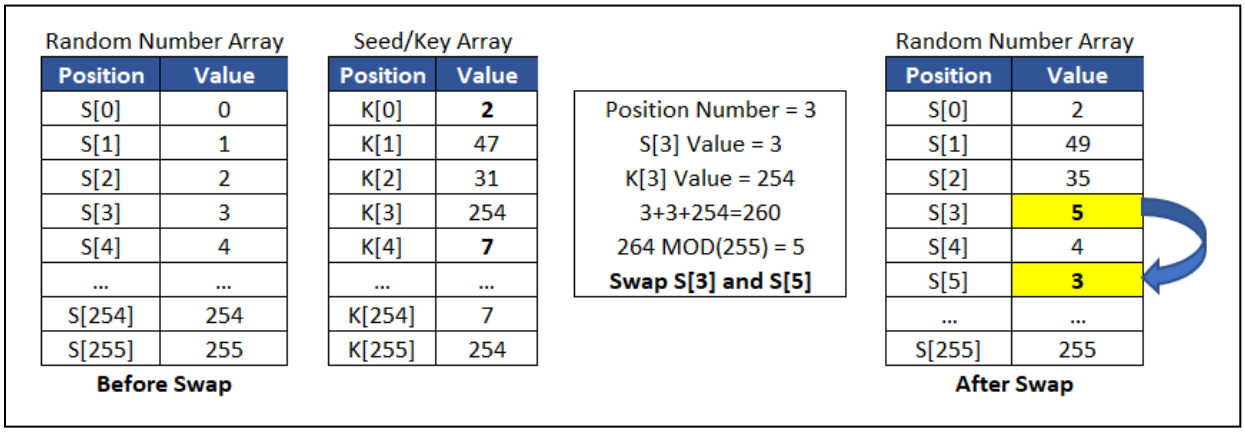
After the K[ ] array is filled the scrambling of the S[ ] array begins. This step is very similar to shuffling a deck of cards, where the order of the cards is changed, but the deck starts and ends with the same cards. That is, no cards are removed and no new cards are added, it's the same cards but 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 all 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. The calculation takes add the position number, the value in the current position, and the value in the corresponding position in the K[ ] array. After adding these three values together the result is MODed by 256 ensuring the result will be between 0 and 255. The result of this calculation is used as the position number for the second number in the S[ ] array to swap with.

For example, let's assume that seed values are 2, 47, 31, 254 and 7, which means the K[ ] array will hold the values shown in the following figure. When it's time to start shuffling the values in the S[ ] array the first element to swap will be S[0] which initially holds the value 0. The calculation to determine the other S[ ] array element to swap with s[0] uses the values shown in the figure. The result of this calculation is 2, which means that the values in S[0] and S[2] will be swapped. This process is repeated for every element in the S[ ] array.



Let's do one more example to demonstrate what happens when the result of the additions is greater than 255. In this case let's calculate the S[ ] element to swap with S[3]. The calculation to determine the other S[ ] array element to swap with s[3] uses the values shown in the figure. The result of the additions 260, but this can't be the final result as there's no array element S[260]. This is why the MOD is done after the additions, to ensure the result is between 0 and 255. In this case the remainder left over after dividing 260 by 255 is 5, which means that the values in S[3] and S[5] will be swapped.



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 different seed/key values are used.

It's important to note that one of the factors in the swapping calculation for each element of the S[ ] array is the value in the corresponding element of the K[ ] array. This means the order of the numbers in the S[ ] array after the swap will be different for each different RC4 seed/key.

RSA PRNG

Once the Key Scheduling algorithm is finished the RC4 cipher starts generating pseudo random numbers using its 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 the RC4 process to understand may be 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 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 seed/key used to generate it. In this case doing the extra shuffle would be wasted effort. However, the additional time required to do the extra shuffle would have a negligible impact for a single message on modern CPUs.

But if the message is longer than 256 characters and the random numbers in the S[ ] array aren't reshuffled, it would result in the same set of random numbers being used in the same order. This repetition 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 that may not be obvious.

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} possible 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. For example, assume the PRNG returns the integer 27 and the plain text character is **T**. The binary

value of 27_{10} is 0001 1011_2 and the binary value for the ASCII character T is 0101 0100_2. The XOR operation combining these two binary numbers is shown below:

```
    0001 1011
XOR 0101 0100
    0100 1111
```

This process can be used to encrypt any type of data as the unencrypted 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 an even deeper understanding of the RC4 cipher, here are the details. 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 the code reads in the encryption key values, which consist of 5 – 256 non-negative integers whose values are between 0 and 255. The code also reads 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 SEED[ ].

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] = SEED[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:

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

The `swap()` function is fairly straightforward, but it needs to store one of the values in a temporary variable as the original value will overwritten during the swap.

```
function swap(S[i], S[j])
      tempVar = S[i]
      S[i] = S[j]
      S[j] = tempVar
```

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 seed/key is no longer involved. It was only used during the initial shuffle of the S[ ] 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 critical 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 another 24-bit value called an Initialization Vector or 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 the attacker can gain access to the key. This makes wireless routers that use WEP extremely vulnerable as 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. It doesn't matter what encryption algorithm is used, 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.

XXX – Need to add details on weaknesses, Look in Aumaason book in RC4 section

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

In any case, researchers have identified several attacks[2][3] 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 stream 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 which was 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 as part of the Internet infrastructure, SSL/TLS which you'll learn about later in the book.

Generally speaking the 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 XORed with the plain text bits to perform the encryption. The thing about these algorithms that is 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 first learn the details of the ChaCha20 cipher as it's the most modern. After this, the details of the Salsa20 implementation will be presented. You can probably skip the Salsa20 description unless you interested in a career specializing in cryptology.

Simplest Explanation of the Salsa20 and ChaCha20 Algorithms

[2] https://www.rc4nomore.com/
[3] https://crashtest-security.com/disable-ssl-rc4/

This section will provide the simplest explanation of the Salsa20 and ChaCha20 algorithms. It will be light on technical details but provides an overview of how the PRNG for these algorithms functions.

Both algorithms start with square matrix that will be used to generate and hold the random numbers. This size of this matrix is 64 bits x 64 bits, or 8 bytes x 8 bytes.

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

Both algorithms then load the matrix with the following pieces of information, they just put it in slightly different places as shown in the following figure.

1. The seed/key numbers. These must be 32 numbers between 0-255. These are shown in the figure as k1-k32.
2. A constant string **expand 32-byte k**
3. A nonce. This is a number that is incremented each time a new message is encrypted.
4. A counter. This is a number that is incremented each time the matrix is changed to generate a new set of random numbers to encrypt bits in the same message. It's always reset to 0 any time a new message is encrypted.

Salsa20

e	x	k1	k2	k3	k4	k5	k6
p	a	k7	k8	k9	k10	k11	k12
k13	k14	n	d	Nonce			
k15	k16		3				
Counter				2	-	k17	k18
				b	y	k19	k20
k21	k22	k23	k24	k25	k26	t	e
k27	k28	k29	k30	k31	k32		k

ChaCha20

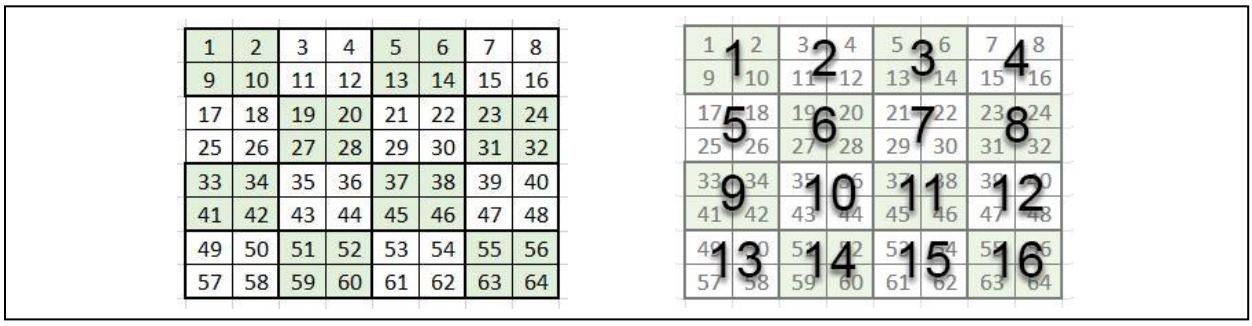
e	x	p	a	n	d		3
2	-	b	y	t	e		k
k1	k2	k3	k4	k5	k6	k7	k8
k9	k10	k11	k12	k13	k14	k15	k16
k17	k18	k19	k20	k21	k22	k23	k24
k25	k26	k27	k28	k29	k30	k31	k32
Counter				Nonce			

It might help to think of these as 64 cards laid out in 8 rows, with 8 cards in each row. Each of the 64 cards has a number on it, ranging from 0 to 255. When the process of encrypting a new message begins, the constant cards will always be set to the same numbers, and the counter cards will all be set to 0. The key cards will hold numbers that you have chosen for the seed/key, and should be different for every message that's encrypted. The nonce cards will be set to 0 the first time a message is encrypted, and then incremented by one each time a new message is encrypted. In the end of this initial setup you'll have a matrix containing 64 numbers.

This is different than the initial setup in RC4 in a few ways:

1. There are 64 numbers instead of 256
2. Some numbers may occur multiple times
3. The seed/key numbers are added to the main matrix
4. It uses the constant, nonce, and counter

The 8-bit numbers in the array cells are then combined, 4 at a time, to create a set of 32-bit numbers. This results in an array of sixteen 32-bit numbers, as shown in the following figure.



These 16 numbers are then changed using a series of calculations. Without getting into too much detail here's how the changes occur. Each number in the array is changed by one of the following actions:

1. The number is replaced with the number that results from adding the number in the current position to a number in a different array position.
2. The number is replaced with the number that results from XORing the number in the current position to a number in a different array position.
3. The bits in the number are rotated a certain number of places right or left.

If we apply the playing card analogy to this process it would be like going through each card on the table and replacing it with a completely different card. The way you determine the new card is by picking a second card, and using the values of the initial card and the second card to determine the replacement card. For example, one of the replacement calculations might add the values of the initial card and the second card to determine the replacement card, but always keeping the same suit of the initial card. If the initial card is a 4 of hearts and the second card is a 5 of any suit, the replacement card would be a 10 of hearts. Or you might have a rule that changes the card suit using a table like the following. The main point to take away is that at the end of the scrambling process there will be 64 completely different cards than what you started with.

Card 1	Card 2	Result
♠	♦	♥
♣	♥	♦
♥	♣	♠
♦	♠	♣
…		
♥	♥	♠

If you want to know exactly how the two number positions are determined, and the exact calculations used to change the numbers, you can look at the detailed explanations later in the chapter. For this simple explanation the point to take away is that all this scrambling is done get a set of random bits. And you might think that after the changes are applied to each number in the array the result would be significantly random. But to ensure that numbers are truly scrambled these changes are applied multiple times following this process:

1. Each number is changed twice, using the process described above using an initial set of number positions. For example, perform the calculation on number in position 1 by combining it with the number in position 9. Do this twice.
2. Change each number as described above twice using a different combination of positions. For example, perform the calculation on number in position 1 by this time combining it with the number in position 10 instead of position 9.
3. Repeat steps 1 and 2 nine more times, for a total of 10 times.

The number of times these changes happen are measured in something called rounds. The problem with getting to rounds from the number of calculations is like figuring out how many tablespoons are in a gallon. In this case the two changes in step 1 count as a round, and the two changes in step 2 count as another round. These 2 rounds happen 10 times, which results in a total of 20 rounds. This is where the 20 part of the Salsa20 and ChaCha20 names comes from.

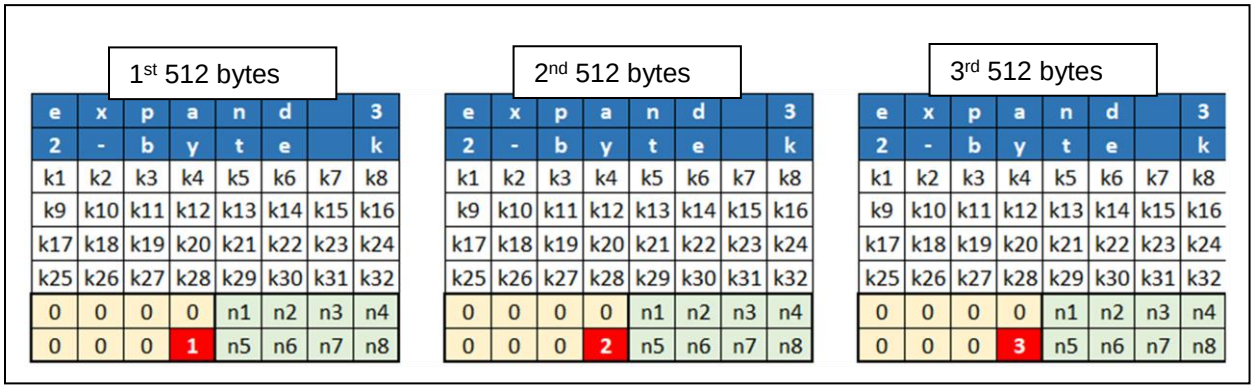
Once all of the rounds of calculations have been completed the numbers from the original matrix are added back in to the numbers in the matrix. This is done to make it impossible for someone to reverse the 20 rounds of calculations and discover the seed/key.

Scrambling the numbers for 20 rounds results in significant changes to all the numbers in the array, and also randomizes the numbers enough to turn this process into a CSPRNG. Note that the Salsa20/ChaCha20 scrambling is much different than the shuffling done in RC4 which didn't change any of the numbers. RC4 changed the order of the numbers, while the Salsa20/ChaCha20 scrambling ends up making significant changes to the numbers, so you end up with a completely different set of numbers than you started with. But despite this difference both cases, in the end both RC4 and Salsa20/ChaCha20 produce a set of random numbers.

Once the 20 rounds of scrambling are completed, the bits from the numbers in the array are combined with the bits from the data being encrypted using the XOR function. The actual encryption is nothing special or different. The special thing about Salsa20 and ChaCha20 are the way their CSPRNGs work.

Another feature of the Salsa20/ChaCha20 PRNGs to look at is how much data can be encrypted with the random bits from the array. The array holds 64 numbers or 512 bits, which means it can be used to encrypt 512 bits of data. If more than 512 bits of data are being encrypted the entire matrix will be reset to the original values with the exception of the counter number portion of the matrix which will be incremented by 1. The following figure shows how the matrix is reloaded with same data as it was initially, with the exception of the counter which is incremented by 1 each time the array is reloaded. It's interesting to note that the way the scrambling works, just changing the counter causes changes to propagate through the entire

matrix and simply incrementing the counter is enough to produce huge changes in the final set of scrambled bits.



e	x	p	a	n	d		3
2	-	b	y	t	e		k
k1	k2	k3	k4	k5	k6	k7	k8
k9	k10	k11	k12	k13	k14	k15	k16
k17	k18	k19	k20	k21	k22	k23	k24
k25	k26	k27	k28	k29	k30	k31	k32
0	0	0	0	n1	n2	n3	n4
0	0	0	1	n5	n6	n7	n8

e	x	p	a	n	d		3
2	-	b	y	t	e		k
k1	k2	k3	k4	k5	k6	k7	k8
k9	k10	k11	k12	k13	k14	k15	k16
k17	k18	k19	k20	k21	k22	k23	k24
k25	k26	k27	k28	k29	k30	k31	k32
0	0	0	0	n1	n2	n3	n4
0	0	0	2	n5	n6	n7	n8

e	x	p	a	n	d		3
2	-	b	y	t	e		k
k1	k2	k3	k4	k5	k6	k7	k8
k9	k10	k11	k12	k13	k14	k15	k16
k17	k18	k19	k20	k21	k22	k23	k24
k25	k26	k27	k28	k29	k30	k31	k32
0	0	0	0	n1	n2	n3	n4
0	0	0	3	n5	n6	n7	n8

After the array is reloaded the 20 rounds of mixing will be repeated to generate another 512 bits. It might seem the change to the counter in a regular predictable way will cause patterns to occur. But the way the numbers are mixed and scrambled causes significant changes and even making a small change like incrementing the counter ensures that the resulting bits will be different and no patterns will be produced. The area of the array set aside for the counter is 64 bits, which means the counter can go as high as 18,446,744,073,709,551,616. This number is large enough that it will never repeat during the encryption of a single message.

Before we end this simple explanation of the Salsa20 and ChaCha20 PRNGs let's look at the two main differences between Salsa20 and ChaCha20 algorithms. The first, which was pointed out above, is the way the different pieces of data are added to the matrix. The second difference is a technical change to the algorithm used when selecting the second number to use when scrambling each number in the array. It's difficult to describe how this was changed without taking a deeper dive into the details of the algorithms, so at this point let's just stick with the general description of the change. If you want to see the details of the change it's included in the detailed descriptions of the algorithms provided below.

The last thing to note about these changes is that Mr. Bernstein made them to increase the performance of his Salsa20 algorithm. After working with Salsa20 for several years he found that these slight tweaks made it possible for ChaCha20 to encrypt messages faster than Salsa20.

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 of these items which includes how they're generated and how they're used.

Key – I'm pretty sure you're familiar with the seed/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. On computer systems this is typically done by getting a number from a system measures true entropy, such as the number of network packets received in a certain time period, etc. This 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. 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 the program runs. The encryption application uses a stored piece of data to track the nonce, so that it can increment the nonce for each new message and guarantee that the same nonce is not used for multiple messages. It may seem like the nonce will be repeated sooner or later, which might create a pattern and 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 1, while the nonce will not be reset. 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 1 at the start of decryption, and then incremented each time a new matrix of key bits is required.

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

Detailed Explanation of the ChaCha20 PRNG

Now let's take a look at the details of Salsa20 and ChaCha20. We'll start by looking at ChaCha20 since it's the streaming algorithm that's currently being used.

As described above ChaCha20 sets up a 64 bit by 64 bit matrix, and uses it to generate 512 random bits. When ChaCha20 initializes the matrix, it adds a 256 bit seed/key, a constant, a nonce and a counter.

During initialization the first thing ChaCha20 places in the matrix is the seed, which is placed in the middle rows. While the seed/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

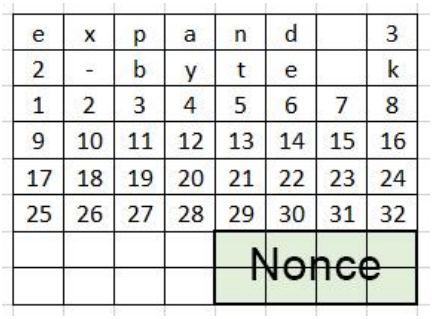
The next section of the table is filled by the bytes for two 32-bit counters. These counters keep track of how many times the table has been used to generate the set of 512 pseudo random bits for a single message. That is, ChaCha20 reuses this matrix every time it needs an additional set of random bits. The following 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 first time the ChaCha20 matrix is loaded the counters will be set to 1. The scrambling of the matrix produces 512 random bits, so if the message being encrypted or decrypted is longer than 512 bits another set of bits must be generated. When a new set of bits is needed the matrix will be reloaded with the same initial values, except the counter, before repeating the scrambling process. Each time the matrix is reset to the initial values for the *same* message the counter is incremented by one. This means to generate the second set of 512 random bits the counter will be set to 2, for the third set of bits it will be set to 3, etc. As pointed out above, the way the scrambling works just changing the counter causes changes to propagate through the entire matrix during the scrambling process, and simply incrementing the counter is enough to produce huge changes in the final set of scrambled bits.

The last technical detail about the counter is that any time the encryption or decryption process is started for a *new* message the counters are reset to 1.

The last 64 bits of the matrix are used to hold something called a nonce. Most definitions of the word nonce say it means something that is only used once which is how it's being used here. However, in Britain, nonce is slang for sex offender, so be careful if you talk cryptography with anyone from Britain! The nonce is very similar to the seed/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, when you hear the term encryption key, which means the information required to encrypt and decrypt a message, in this case it means both the nonce and the seed/key.



The difference between the nonce and the seed/key is how they're generated. Generally speaking, the seed/key is a set of random numbers which the OS produces by grabbing numbers from some random process on the computer such as CPU usage in the last few milliseconds, while the application itself generates the nonce, typically by using a counter so that it can ensure that the counter is not repeated. For example, the very first time a program on a computer uses ChaCha20 to encrypt a message it sets the nonce to 1. 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 2, which will be the nonce number used to encrypt the entire second message.

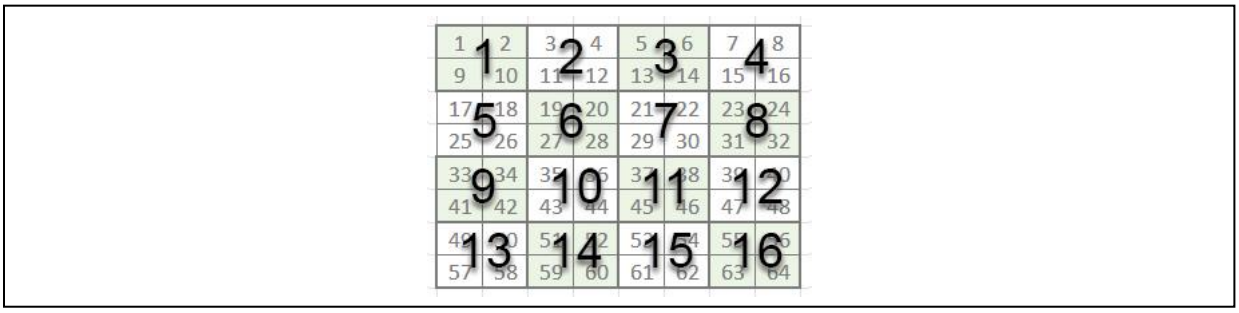
Using a nonce prevents problems that would occur if for some reason the OS ends up returning the same numbers for the seed/key. If the OS does return the same seed/key numbers, without the nonce you'd end up encrypting the messages using the same seed/key. Remember that using the same key for multiple 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 somehow the same seed/key is used.

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 will really help you visualize 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 from the original matrix. Since each cell of the original matrix contains 8 bits of binary data, each of these new sections will hold 32 bits of binary data. I'm going to call these new 32-bit sections blocks to distinguish them from the original 8-bit cells. This will result in a new 4x4 matrix, as

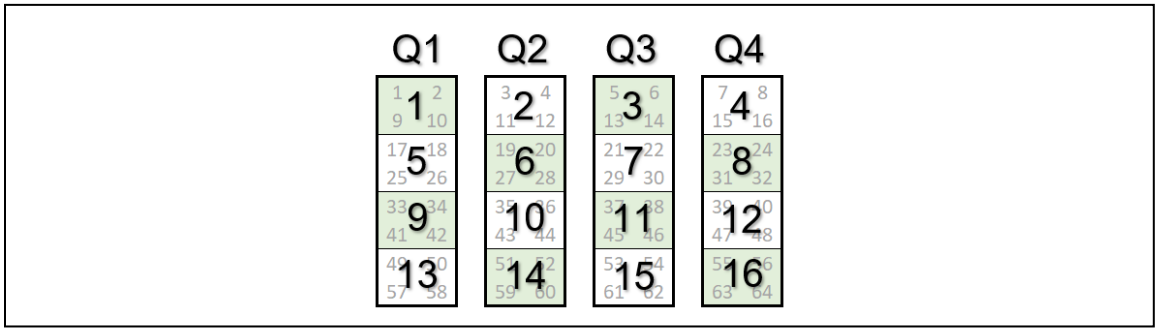
shown in the following figure. In the figure the original cell numbers are displayed in the small font and the new block numbers are in the large font. Remember that the numbers shown aren't what's being stored in the cells, they're just labels to help us keep things straight.



In the new matrix the data in block 1 will be the bits from cells 1, 2, 9 and 10 from the original 8x8 matrix; the data in block 2 of the new matrix will be the bits from cells 3, 4, 11 and 12 of the original 8x8 matrix, etc.

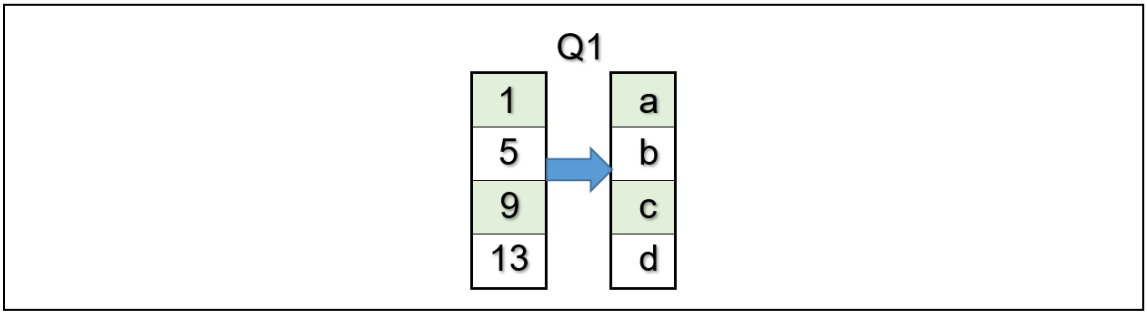
At this point ChaCha20 performs10 repetitive rounds of scrambling. The steps in this scrambling are:

A. Sub-divide the 4x4 matrix into groups of 4 blocks. These groups are called quarters because the block is a quarter of the original matrix. For example, one of the ways ChaCha20 divides the matrix into quarters would be to break it into 4 columns.



B. The 32-bit numbers in each quarter 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. For example, using the blocks in the first column or Quarter 1, the number in

block 1 becomes **a**, the number in block 5 is mapped to the letter **b**, the number in block 9 is mapped to **c**, and the number in block 13 is assigned the letter **d**.



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

```
a = a+b
d = d XOR a
d <<<= 16
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
```

(This shifts the bits in the number d 16 places to the left)

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.

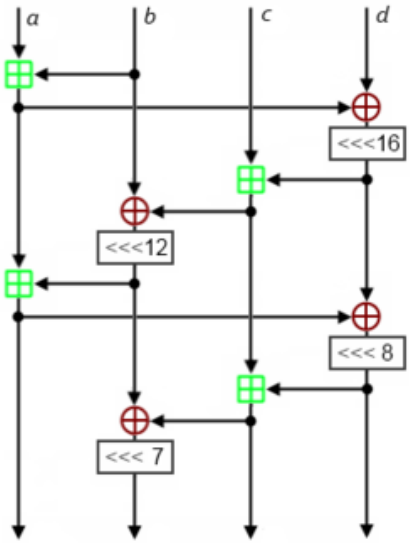
Take for example $\mathbf{11}000011_2$ <<<= 2 This says to shift the bits two places to the left. Instead of just shoving the left two most bits off the left end, they are wrapped back around and placed on the right, resulting in: $000011\mathbf{11}_2$. Note that the number in this

example is an 8 bit number, while the numbers being used in the ChaCha20 algorithm are 20 bit numbers. Even though the example has fewer bits the concept of shifting or rotating bits will be the same for 32 bit numbers.

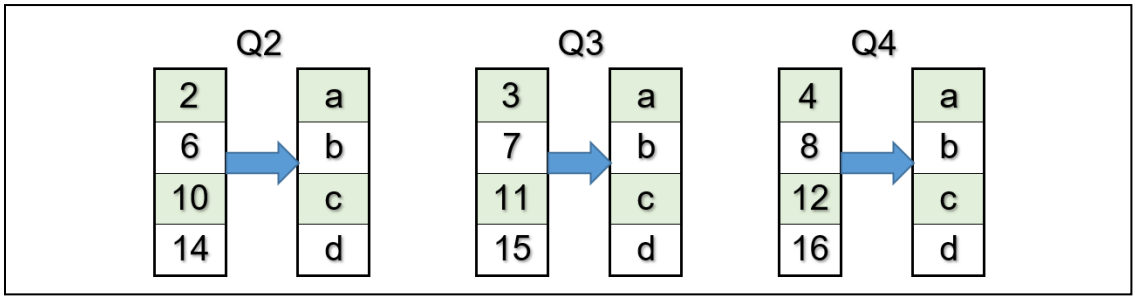
The C/C++ family of programming languages use what they call shorthand operators for the addition and XOR operations. The addition shorthand is `+=`, so `a+= b` is the same as `a = a+b`. The shorthand for the XOR operation is `^=`, so `d ^= a` is the same as `d = d XOR a`. Using the shorthand operators 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.



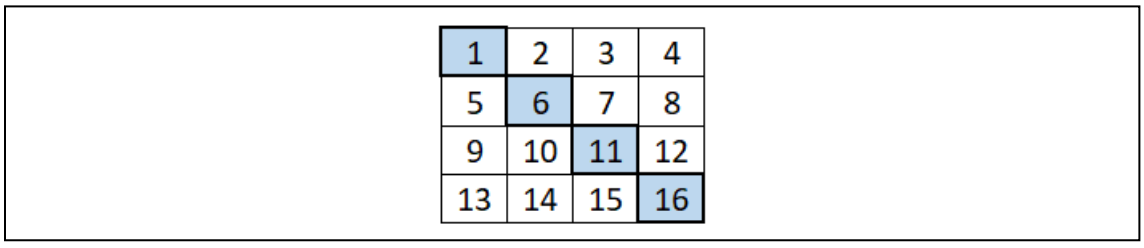
It's important to note that this scrambling happens to the numbers in all four quarters or columns. Each time a new column is scrambled the top cell is assigned to **a**, the second cell down is assigned to **b**, the third cell down to **c**, and the bottom cell is assigned to **d**. the following figure shows how the block to letter mappings are done for the remaining quarters or columns.



C. After the scrambling operations are performed on all 4 quarters, they're repeated once again. That is, all the operations in Step B are repeated on each quarter.

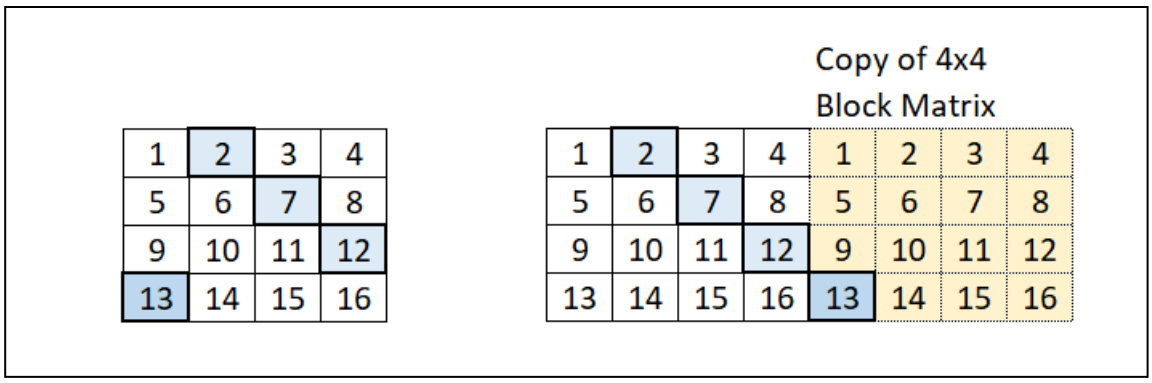
D. After the column scrambling is finished the 16 block matrix is sub-divided again, but this time into 4 diagonal sections instead of into columns. If you just look at the algorithm for doing this diagonal sub-division it can be confusing, but when you see a picture of how it's done it's pretty straightforward.

Here's the first diagonal. It consists of cells 1, 6, 11, and 16.

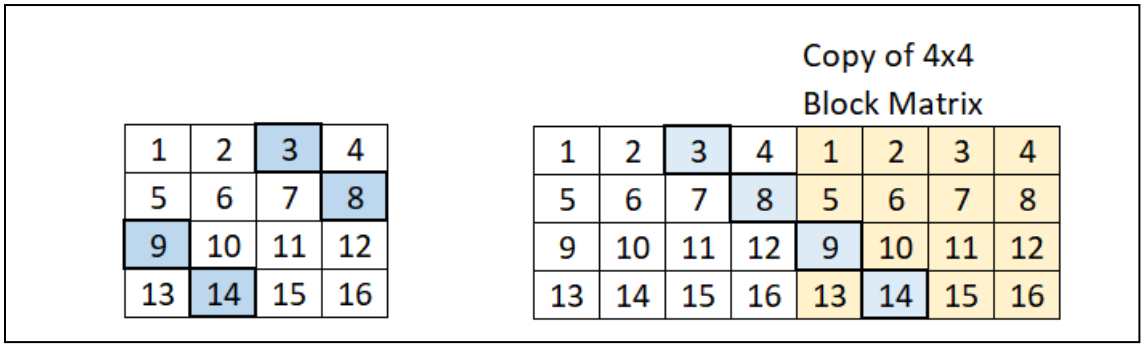


The second diagonal has to wrap around to get the 4th cell. It consists of cells 2, 7, 12 and 13. The diagonal blocks and the wrapping used to create the quarter block might be easier to see if we place a duplicate of the table next to the original table as shown in

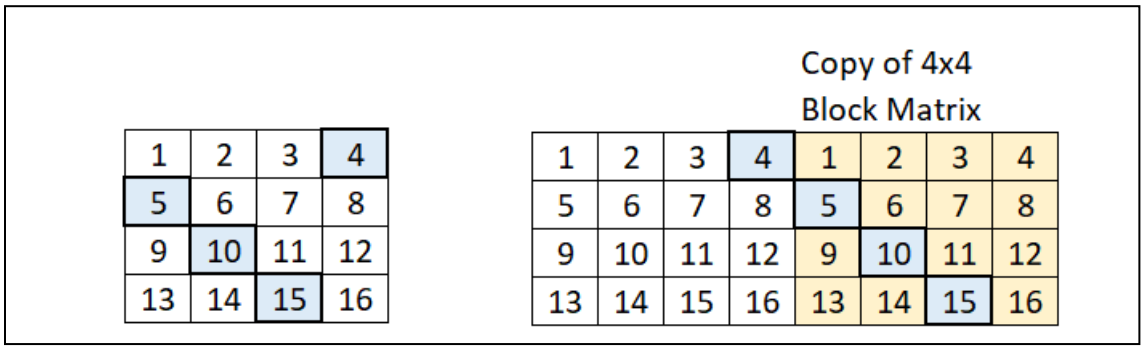
the following figure where the duplicate blocks are colored yellow. Note that the numbers in the blocks aren't actually duplicated, it's all done using some math.



The third diagonal also has to wrap around to get the last two blocks. It consists of the blocks 3, 12, 17 and 26. The following figure shows the blocks used in the diagonal from the single table on the left, and the same blocks using a duplicated table that makes it easier to see the diagonal on the right.



The fourth diagonal must also wrap around to get the last three numbers. It consists of the cells 4, 5, 10 and 15. The following figure shows the blocks used in the diagonal from the single table on the left, and the same blocks using a duplicated table that makes it easier to see the diagonal on the right..



E. After each quarter is set, the four numbers in the diagonals are scrambled using the same 12 formulas from Step B used to scramble the numbers in the columns. This scrambling happens to the numbers in all 4 diagonals. As with the columns, the numbers in each diagonal quarter are scrambled twice using the operations in Step B.

F. Performing the scrambling operations on all 4 quarters twice is called a round. At this point one round of scrambling has been done on each of the 4 column quarter groups, and another round has been done on each of the 4 diagonal quarter 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.

G. 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.

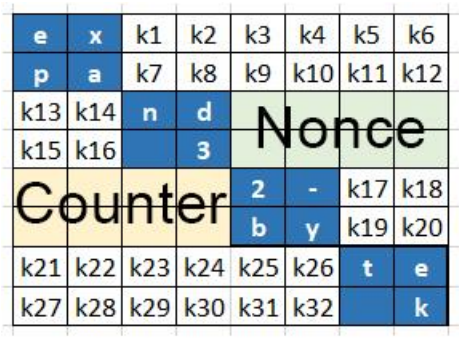
When all the scrambling is finished the matrix holds 512 bits, called a keystream. This is the set of bits which can be combined with the plain text using the XOR function to encrypt the message, or combined with the cipher text using the XOR function to decrypt the message.

Since the matrix only holds 512 bits, it can only be used to encrypt or decrypt 512 bits. If additional random bits are required 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.

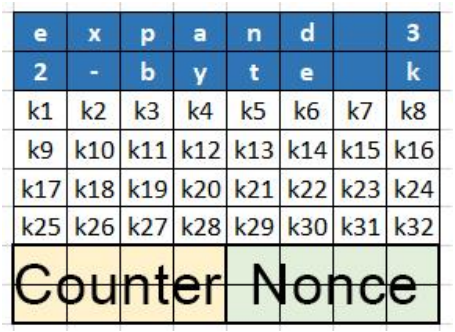
Differences Between Salsa20 and ChaCha20

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.

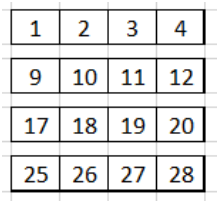


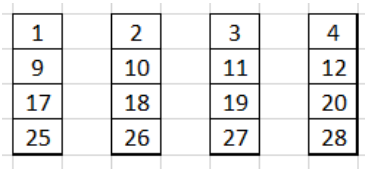
Salsa20



ChaCha20

The only other significant difference is that during the rounds Salsa20 first divides the matrix into rows instead of columns.





Usage

The last thing to discuss about RC4 and Salsa20/ChaCha20 is why they're important. To understand their importance, you'll need a little background information on Secure Sockets Layer (SSL) and Transport Layer Security (TLS) which are critical components of the Internet infrastructure. You'll learn the details of SSL and TLS in a later chapter, but for now you need to know enough to understand the role that RC4 and ChaCha20 play.

When the Internet first started the goal was simply to create network connections over long distances. The TCP/IP protocol was developed and adopted as the standard, but it had no built-in security, just ways to ensure that packets could be sent from one computer to another even if the packets had to be routed across different networks and network connections. For many years security was an afterthought, and if it was required by an application it had to be implemented in code in that specific application. When the web and HTTP came around, and the Internet began allowing businesses and commercial web sites it started to become glaringly apparent the need for secure communications started to become apparent. This led to the development of HTTPS and Secure Sockets Layer, which implemented a suite of various encryption and authentication algorithms. The encryption and authentication programs were initially built into the code for web servers and browsers, but eventually became part of the standard network stack in most Operating Systems, so they could be easily accessed by any application that requires a secure network connection. The first code that provided the suite of encryption and authentication software was called Secure Sockets Layer (SSL) and the first symmetric cipher that was included as part of the SSL suite was RC4.

SSL went through a few iterations as different components in the encryption and authentication algorithms were improved or replaced. During one of the SSL revisions some fairly significant changes were made and the decision was made to change the name from SSL to Transport Layer Security (TLS) in an attempt to make it clear that these were major changes. RC4

continued to be included as an option for a symmetric stream cipher in SSL 3.0 and of TLS versions 1.0, 1.1, and 1.2. This basically meant that RC4 was available as an option through 2020 as part of TLS 1.1. ChaCha20 became available as an option in TLS 1.2, which was released in 2008, and is currently the only approved option for a symmetric stream cipher. The following table summarizes information from Wikipedia on SSL/TLS. It shows the year that each version of SSL/TLS was released as well as the year it was deprecated, which means support was dropped and it strongly urged that the version be removed from use. Any boxes in red mean that the version should no longer be used, and should be replaced with a newer version. Any boxes in green mean that the version is currently approved for use.

	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3
Published	1995	1996	1999	2006	**2008**	**2018**
Deprecated	2011	2015	2020	2020		
Stream Ciphers						
ChaCha20					✓	✓
RC4	✓	✓	✓	✓	✓	

Summary

You've been presented with a little new terminology, and a lot of details about the RC4 and Salsa20/ChaCha20 ciphers. I realize that absorbing the details of the ciphers can be a little confusing, especially if this is the first time you've seen them. 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 seed/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 256-bit seed/key is called a seed when the bits are used in the matrix used to generate the random bits. But when we talk about information that the recipient needs to decrypt the message we call these same bits the key. This can be terribly confusing, as we also call the random bits produced by scrambling the matrix the key bits, since they are XORed with the plain text to produce the cipher text. The way to tell which meaning key should use, you must look at the context.

10. The recipient of a message encrypted with Salsa20/ChaCha20 requires the 256-seed/key and the nonce to decrypt it.

11. A nonce is like a key because it's needed to generate the bits to encrypt and decrypt a message. But instead of being completely random like the seed/key bits the nonce is a sequential number generated and tracked by the application.

12. For many years RC4 was used in SSL/TLS, which is the main component of the Internet that provides a suite of applications that can be used to provide secure network connections. RC4 has been replaced by ChaCha20.