Confusion and Diffusion

"The two basic techniques for obscuring the redundancies in a plaintext message," ...
"two properties of the operation of a secure cipher"
- Schneier, 1996; Shannon; "Communication Theory of Secrecy Systems" 1949.

Confusion (substitution)
- obscures the relationship between the plaintext, the ciphertext, and the key to thwart finding redundancies, statistical relationships and patterns that are exploited by linear and differential cryptanalysis. Good confusion makes ciphertext too complicated so powerful cryptanalytic tools won't work. The easiest method is substitution

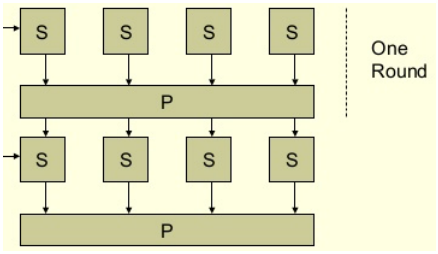
The simple Caesar Cipher, in which every identical letter of plaintext is substituted by the character alphabetically three to the right modulo 26, is easy to break since the ciphertext is a rotation of the plaintext alphabet and not an arbitrary substitution. In ROT13 (not intended for security), every letter is rotated 13 places (A is replaced by N, etc). Encrypting a file twice with ROT13 restores the original file. P = ROT13 (ROT13 (P))

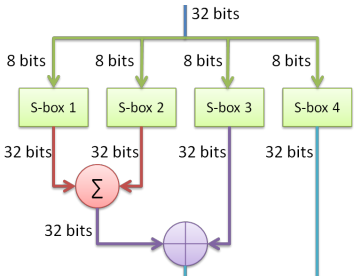
In the secure modern ciphers a long block of plaintext is substituted for a different block of ciphertext, and the mechanics of the substitution change with each bit in the plaintext or key. The famous German Enigma code in World War II was a complex substitution cipher

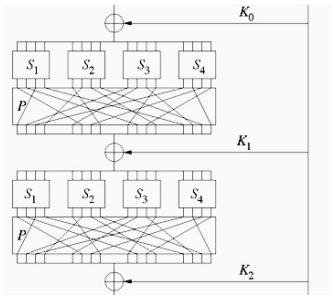
Diffusion (transposition/ permutation).
- dissipates the redundancy or other influence of individual plaintext or key bits over as much of the ciphertext as possible. The simplest way to cause diffusion is through transposition (aka permutation). The old columnar transposition simply rearranges the letters of the plaintext, but modern permutation ciphers use other forms of diffusion that can diffuse parts of the message throughout the entire message.
Generally, diffusion alone is easily cracked (double transposition is good but not enough)

- Confusion provides a secure encryption component. Adding diffusion makes cryptanalysis harder
- Stream ciphers generally rely on confusion alone, although some feedback schemes add diffusion.
- Block algorithms generally use both confusion and diffusion.

Product Ciphers- (putting components together efficiently)
Product ciphers use alternating substitution and transposition phases (rounds) to achieve both confusion and diffusion respectively. When we diagram out the operations of a cipher, it's components are often arranged in a Feistel function/network, which first divides the input of bits into chunks, which have their own paths through the cipher's algorithm. Down the path, confusion is performed/contained in substitution S-boxes, while diffusion/permutation is performed/contained in P-boxes. Each "box" uses a lookup table for the corresponding output values, to pass to the next function, which could be another round through these boxes, with possibly an XOR or other function in between them. The Feistel network containing all this may also be referred to as a substitution-permutation network, or SP network, and it wraps the functions together in a way that can be used to both encrypt and decrypt.







These structures don't only provide strong encryption, but break down tasks into realistic memory chunks to make things possible. A single key-dependent lookup table of 64 bits of plaintext to 64 bits of ciphertext would be strong algorithm, but the table uses 10^20 bytes (100 Exabytes) - large lookup tables require lots of memory to implement. The whole point of block cipher design is to create something that looks like a large lookup table, but with much smaller memory requirements.

[preamble: Stream vs block - A Bogus Premise]
The distinction between block and bit stream ciphers is not always clear-cut: in some modes of operation, a block cipher primitive is used so it acts effectively as a stream cipher, which is how CFB and OFB work
This is precisely where information developed for certification tests makes people not understand the way this all works: it makes an artificial brick wall between the two types as the basis, an oversimplification so that the questions can be answered on the test

Algorithm Types: block and stream ciphers
Two types of symmetric algorithms: block and (bit) stream ciphers

"Block ciphers operate on data with a fixed transformation on large blocks of plaintext data; (while) stream ciphers operate with a time-varying transformation on individual plaintext digits." [Rueppel]

- Stream ciphers generally rely on confusion alone, although some feedback schemes add diffusion.
- Block algorithms generally use both confusion and diffusion.

Although block and stream ciphers are very different, block ciphers can be implemented as stream ciphers and stream ciphers can be implemented as block ciphers.
Main differences are in implementation.
- Stream ciphers can be more suitable for hardware because the device sees individual bits.
- Block ciphers can be easier to implement in software, often avoiding time-consuming bit-by-bit operations
- Encrypting the link between the keyboard and the CPU in software sounds ideal for a stream, but generally an encryption block could be at least the width of the data bus.

In the real world, block ciphers seem to be more general (i.e., they can be used in any of the four modes) and stream ciphers seem to be easier to analyze mathematically.

Block ciphers:
- "Operate on blocks of plaintext, ciphertext is usually 64 bits but sometimes longer
- "Same plaintext block will always encrypt to the same ciphertext block, using the same key

Stream ciphers:
- "Streams of plaintext or ciphertext one bit or byte (sometimes even one 32-bit word) at a time"
- "The same plaintext bit or byte will encrypt to a different bit or byte every time it is encrypted"

RC4 is the most used and well-known
- Was an RSA trade secret 1987, leaked to general use in 1994 (aka ARC4 for “alleged” due to trademark.
- RC for "Rivest Cipher" also informally stands for "Ron's Code"
- Used in Wired Equivalent Privacy (WEP), BitTorrent, Microsoft's Remote Desktop Protocol (RDP)
- The prevalent example is WEP, deprecated in 2004 due to its implementation of RC4.
- Used in SSL(1995)/TLS(1999) and WEP. Was chosen for speed and simplicity - usage not advised
- Use for TLS was already frowned upon, but in Feb 2015 was finally prohibited in RFC 7465.
- Pseudorandom generator used. Proposed new random number generators are often compared to RC4's.
- A few OS's use RC4 for random number generation
- Several attacks on RC4 are able to distinguish its output from a random sequence.
- Other official variants/ versions includie RC2, RC5, RC6 (RC4 prevails)
- In 2014 Ron Rivest introduced an updated redesign called Spritz, which is slower and not adopted much
- Other attempts to strengthen RC4 include RC4A, VMPC, and RC4+

- Attacks on RC4 include Klein, Royal Holloway, Bar-mitzvah (often WEP-implementation-specific)
- Fluhrer, Mantin and Shamir attack specifically exploited the first 1024 bits of the key
- Numerous Occurrence MOnitoring & Recovery Exploit (NOMORE) attack against TLS and WPA

Other Stream ciphers: A5/1, A5/2, Chameleon, FISH, Helix, ISAAC, MUGI, Panama, Phelix, Pike, SEAL, SOBER, SOBER-128 and WAKE. Comparative table: https://en.wikipedia.org/wiki/Stream_cipher

Block Cipher Confidentiality Modes (of Operation) Part 1

- ECB, CBC, OFB, and CFB date back to 1981, specified in FIPS 81, DES Modes of Operation
- "Usually combines the basic cipher, some sort of feedback, and some simple operations" (Schnier)
- "Operations are simple because the security is a function of the underlying cipher and not the mode"
- The mode "should not compromise the security of the underlying algorithm"
- 2001, NIST added CTR mode in SP800-38A
- 2010, NIST Addendum to SP800-38A - Outlines CBC-CS1, -CS2, and -CS3 (aka CTS "CipherText Stealing mode")
- Other confidentiality modes exist, not approved by NIST.

Basic Block Cipher modes (ECB and CBC)

Electronic Code Book (ECB):

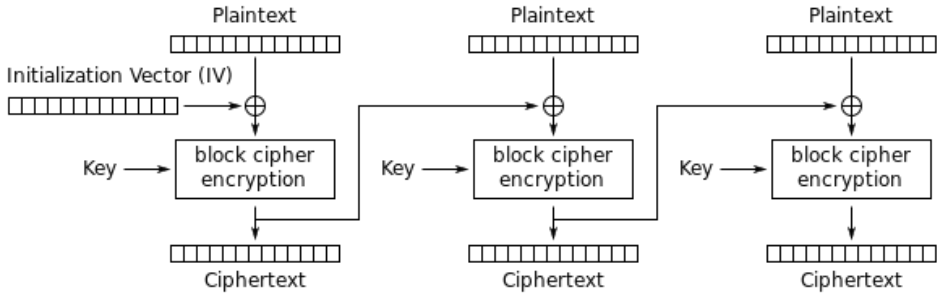
Easy, fast, but also weakest. Weak algorithm and is vulnerable to block replay attacks. Very flawed mode- avoid. Only suitable for encrypting short and random data, like other keys. These uses aren't affected by ECB's problems.

- Each block is encrypted by itself. More than one message can be encrypted with the same key.
- File doesn't have to be encrypted linearly (do 10 blocks at the end, then the start, etc.)
- Each occurrence of a particular word is encrypted exactly the same*
- Two identical plaintext blocks will always generate the same ciphertext block.
- Same plaintext block will always encrypt to same ciphertext - susceptible to brute-force; deletion and insertion
- Plaintext patterns not concealed; input to block cipher not randomized (same as plaintext)
- Plaintext is easy to manipulate, blocks can be removed, repeated, or interchanged.
- Padding the plaintext makes it fit ciphertext fixed-block size (usually 64-bit in ECB)
- Processing is parallelizable, but preprocessing is not possible.
- A ciphertext error affects one full block of plaintext. Synchronization error is unrecoverable.
- If a ciphertext bit is accidentally lost or added, all subsequent ciphertext will decrypt incorrectly unless there is some kind of frame structure to realign the block boundaries
- **If a DB is encrypted with ECB, then any record can be added, deleted, encrypted, or decrypted independently of any other record, assuming that a record consists of a discrete number of encryption blocks*

Cipher Block Chaining (CBC):

- Invented by IBM in 1976
- encryption of each block depends on all the previous blocks (a feedback mechanism)
- plaintext block is XORed with the previous ciphertext block, before encrypted with algorithm and key
- after plaintext block is encrypted, resulting ciphertext is spit out and also piped to the next block's process

The first block doesn't have a previous block to use, so an IV the same size as the plaintext block kicks it off.



Decrypting is similar- ciphertext decrypted normally and also saved in feedback register; next block decrypted, XORed with the feedback register, etc.

Some messages have a common header: a letterhead, or a "From" line, or whatever. While block replay would still be impossible, this identical beginning might give a cryptanalyst some useful information. An initialization vector (IV) throws some random data as the first block; helps make identical plaintext messages encrypt to different ciphertext messages

- sometimes it's just a timestamp, should be unique for each message encrypted with the same key, it is not required.

When the receiver decrypts this block, it's used to fill the feedback register.

The IV need not be secret in CBC; it can be transmitted in the clear with the ciphertext. If this seems wrong, consider the following argument. Assume that we have a message of several blocks: $B_1, B_2, ..., B_i$. B_1 is encrypted with the IV. B_2 is encrypted using the ciphertext of B_1 as the IV. B_3 is encrypted using the ciphertext of B_2 as the IV, and so on. So, if there are n blocks, there are n-1 exposed IVs, even if the original IV is kept secret. So there's no reason to keep the IV secret; the IV is just a dummy ciphertext block- you can think of it as B_0 to start the chaining.

- Easy and fast; best for encrypting files; Almost always the best choice if application is software-based
- More than one message can be encrypted with the same key.

- Plaintext patterns concealed by XORing with previous ciphertext block.
- An IV is used to encrypt the first block, since there is no "previous ciphertext block"

- Processes 64-bit blocks; if last block does not reach the block boundary, it is filled with padding.
- Ciphertext is up to one block longer than the plaintext, not counting the IV.
- A ciphertext error affects one full block of plaintext and the corresponding bit in the next block.
- Almost never synchronization errors, a synchronization error is unrecoverable.
- No preprocessing. Encryptions not parallelizable; decryption is parallelizable and has a random-access property.
- Plaintext is somewhat difficult to manipulate; blocks can be removed from the beginning and end of the message, bits of the first block can be changed, and repetition allows some controlled changes.

The Propagating Cipher Block Chaining (P-CBC) variant of CBC provides error-checking; integrity check. Kerberos 4 used it but it was found to be flawed and dismissed.

Padding in ECB and CBC

ECB requires 64-bit blocks. Most messages don't divide neatly into 64-bit (or whatever size) encryption blocks; there is usually a short block at the end. Padding is the way to deal with this problem.

- Pad the last block with some regular pattern- zeros, ones, alternating ones and zeros- to make it a complete block.
- If you need to delete the padding after decryption, add the number of padding bytes as the last byte of the last block.

Example: assume the block size is 64 bits and the last block consists of 3 bytes (24 bits).
Five bytes of padding are required to make the last block at 64 bits; add 4 bytes of padding and a final byte with the number 5. On decryption, it is understood to delete the last 5 bytes of the block.

For this method to work correctly, every message must be padded.
Even if the plaintext ends on a block boundary, you have to pad one complete block.
Otherwise, you can use an end-of-file character to denote the final plaintext byte, and then pad after that character.

Padding in CBC works like ECB, but in sometimes the ciphertext has to be exactly the same size as the plaintext.
Suppose a plaintext file has to be encrypted and then replaced in the exact same memory location.
In this case, you have to encrypt the last short block differently.
Assume the last block has J bits. After encrypting the last full block, encrypt the ciphertext again, select the left-most J bits of the encrypted ciphertext, and XOR that with the short block to generate the ciphertext.

The weakness here is that while Mallory cannot recover the last plaintext block, he can change it systematically by changing individual bits in the ciphertext. If the last few bits of the ciphertext contain essential information, this is a weakness. If the last bits simply contain housekeeping information, it isn't a problem.
Ciphertext stealing is a better way . Pn-1 is the last full plaintext block, and Pn is the final, short, plaintext block. Cn-1 is the last full ciphertext block, and Cn is the final, short, ciphertext block. C' is just an
intermediate result and is not part of the transmitted ciphertext. The benefit of this method is that all the bits of the plaintext message go through the encryption algorithm.

CBC Error Propagation

CBC mode can be characterized as "feedback of the ciphertext at the encryption end and feedforward of the ciphertext at the decryption end". A single bit error in a plaintext block will affect that ciphertext block and all subsequent ciphertext blocks. This isn't significant because decryption will reverse that effect, and the recovered plaintext will have the same single error.

Ciphertext errors are more common. They can easily result from a noisy communications path or a malfunction in the storage medium. In CBC mode, a single-bit error in the ciphertext affects one block and one bit of the recovered

plaintext. The block containing the error is completely garbled. The subsequent block has a 1-bit error in the same bit position as the error.

Error extension: property of taking a small ciphertext error and converting it into a large plaintext error.
Blocks after the second are not affected by the error, so CBC mode is self-recovering.
Two blocks are affected, but the system recovers and continues to work correctly for all subsequent blocks.

CBC is an example of a block cipher being used in a self-synchronizing manner, but only at the block level.

While CBC mode recovers quickly from bit errors, it doesn't recover at all from synchronization errors.

If a bit is added or lost from the ciphertext stream, then all subsequent blocks are shifted one bit out of position and decryption will generate garbage indefinitely.

Any cryptosystem that uses CBC mode must ensure that the block structure remains intact, either by framing or by storing data in multiple-block-sized chunks.

Structural Vulnerabilities of CBC:

Since a ciphertext block affects the following block in a simple way:
1. Someone can add blocks to the end of an encrypted message without being detected.
 - it will probably decrypt to gibberish, but in some situations this is undesirable.
Solution: structure your plaintext so that you know where the message ends and can detect the addition of extra blocks.

2. Someone can alter a ciphertext block to introduce controlled changes in the following decrypted plaintext block
 - Toggle a single ciphertext bit, the entire block will decrypt incorrectly, but the following block will have a 1-bit error in the corresponding bit position. There are situations where this is desirable.
 - Solution: the entire plaintext should include some kind of controlled redundancy or authentication.

3. (Hypothetical) Although plaintext patterns are concealed by chaining, very long messages will still have patterns. The birthday paradox predicts that there will be identical blocks after $2^{m/2}$ blocks, where m is the block size. For a 64-bit block size, that's about 34 gigabytes. A message has to be pretty long before this is a problem.

Stream ciphers

- Stream of plaintext is processed with a corresponding bit of a pseudorandom cipher one bit at a time
- Fast, require less overhead - one bit at a time perfect for hardware. Output ciphertext is same size as plaintext.
- Good where plaintext quantities are of unknowable length (like a wireless connection)
- Fewer errors, but no guarantee of message integrity;
- Secrecy, but no authentication (traffic can be tampered with, impersonated by MitM)
- As each bit's encryption depends on the state of the keystream generator, the term "state cipher" is also used.
- With binary digits being added to the keystream there is also the name "binary additive stream cipher"

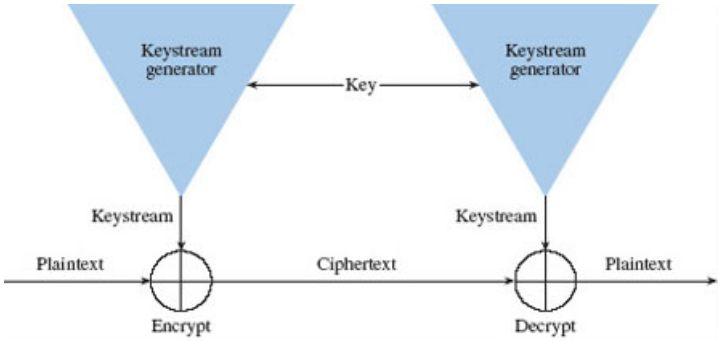
With stream ciphers the fault is that they try to do something proven to work, but completely the wrong way!
- Based on the one-time pad (OTP, Vernam cipher); unbreakable one-time pre-shared key the same size as plaintext
- OTP uses a keystream of truly random digits XORed with plaintext digits one at a time to produce ciphertext
- OTP also requires keystream at least the length of plaintext that will not be used more than once.

A stream cipher fails in 3 ways - uses a key of limited length; repeats the keystream; it's pseudorandom- not random.

Best practices with stream ciphers
- Remember that most stream ciphers provide privacy, but no authenticity: messages in transit can still be modified
- Remember that information can be in certificates may not break the cipher, yet can reveal other weaknesses.
- Never reuse the same keystream twice (generally means a different nonce or key be supplied each time
- Use a method that discards the first few bytes of the keystream
- All keystreams (except one-time pad) are "periodic" -they repeat at some point. The longer this period is, the better
- Although keystream generators work with pseudorandom seeds, the closer to random you can get the better
- Avoid weak keys: as much as possible the keystream should be free of any patterns or detectable relationships that correspond to keys or nonces. The ideal is that transmissions would be indistinguishable from random noise.

Synchronous and Self-Synchronizing: Two categories of Stream Ciphers

Both primary types of stream ciphers are easiest explained by describing the synchronous stream cipher first.
- In a synchronous cipher, a keystream generator creates a keystream from the key (aka "running key"), then the keystream is XORed with the plaintext, and the ciphertext is spit out.
- A generator has an internal state, which is basically which part of the key is used at that moment to make the keystream. Two keystream generators with the same key and same internal state will produce the same keystream.
- On the decoding end, a keystream in the same state generates the identical keystream, then XORs it with the ciphertext to produce the plaintext.



- A stream cipher is defined as *synchronous* simply by the fact that the keystream is generated without the use of either the plaintext or the ciphertext- meaning the keystream is ONLY a function of the key.
- In a synchronous stream cipher both the sender and receiver must be synchronized in that each must be at exactly the same position in their shared key. Any loss or insertion of bits means that they must resynchronize. Big problem since it can force repeating the keystream!. Luckily, errors usually return a corrupted bit, with no interruption.

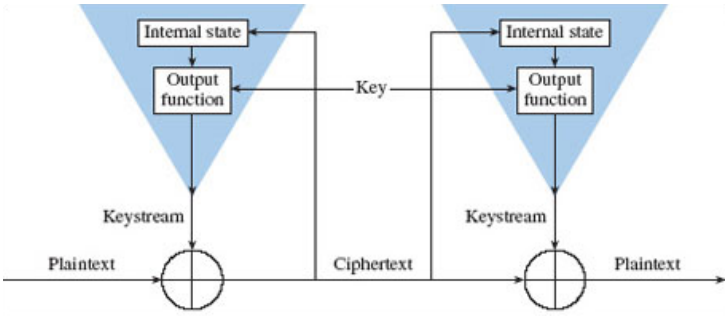
"Synchronous" stream cipher just means that the keystream generation is independent of the plaintext and ciphertext

A few problems with synchronous stream ciphers:
- Have some ciphertext and it's corresponding plaintext? Just XOR both and you get the keystream. Decrypt the rest.
- Have two ciphertexts from the same keystream? XOR both and you have 2 plaintexts XOR'd together. XOR that with one of the ciphertexts and you can get the keystream. Rotating keys can help minimize this vulnerability.
- Even though a loss or addition of bits requires resynchronization, a corrupted or changed bit will not propagate to other bits, and may not even be detected! Useful when transmission error rate is high, but bad for error detection.
- Also a huge attack vector - a change in a ciphertext digit, can make possible predictable changes to corresponding plaintext bit; flipping a bit in the ciphertext causes the same bit to be flipped in the plaintext.

Self-Synchronizing Stream Ciphers

In a synchronous stream's keystream generator, we had an internal state working with the key put into the output function to spit out the keystream. Simultaneously, a next-state function took the old internal state to come up with the

next one to use. Instead of setting up the internal state like that, *self-synchronizing* stream ciphers grab the ciphertext that was just generated, and plug it into making the next internal state



In a *self-synchronizing* (or *asynchronous*) stream cipher, the keystream is generated as a function of the key and a fixed number of previous ciphertext bits.

Since the internal state is derived from the previous ciphertext bits, synchronization is automatic after *x* bits have been received. Good implementations start things off with a random header to mark the length of ciphertext bits to use

Problems with self-synchronizing stream ciphers:

Error propagation - for every bit of garbled transmission in the ciphertext, there will be corresponding length of bits decrypted incorrectly until the garbled bits get out of the internal state of the keystream generator.

Vulnerable to playback attack - if the same key is used in a stream, you can play back packet-captured traffic and after resynchronizing into the internal state it will decrypt normally. Unless timestamps are used the other end wouldn't be aware of a replay- (imagine repeatedly replaying traffic resulting in granting bank credits).

Finally, if the stream cipher utilizes plaintext in the keystream generation, it is called *nonsynchronous*

In the military, synchronous stream ciphers are also called "Key Auto-Key" (KAK), while self-synchronous are called "Ciphertext Autokey" (CTAK).