OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Secure Programming

Symmetric Encryption

Prof. Dr. Christoph Skornia
christoph.skornia@hs-regensburg.de

Fundamentals of encryption

Definition: A cryptographic cipher is defined as a

$$(\mathcal{M}, \mathcal{C}, \mathcal{K}_E, \mathcal{K}_D, E, D)$$

with

1. $\mathcal{M}$: Message Space
2. $\mathcal{C}$: Cryptogram Space
3. $\mathcal{K}_E$: Encryption-Keyspace
4. $\mathcal{K}_D$: Decryption-Keyspace and a function $f : \mathcal{K}_E \longrightarrow \mathcal{K}_D$
5. The injective encryption function $E : \mathcal{M} \times \mathcal{K}_E \longrightarrow \mathcal{C}$
6. The decryption function $C : \mathcal{C} \times \mathcal{K}_D \longrightarrow \mathcal{M}$

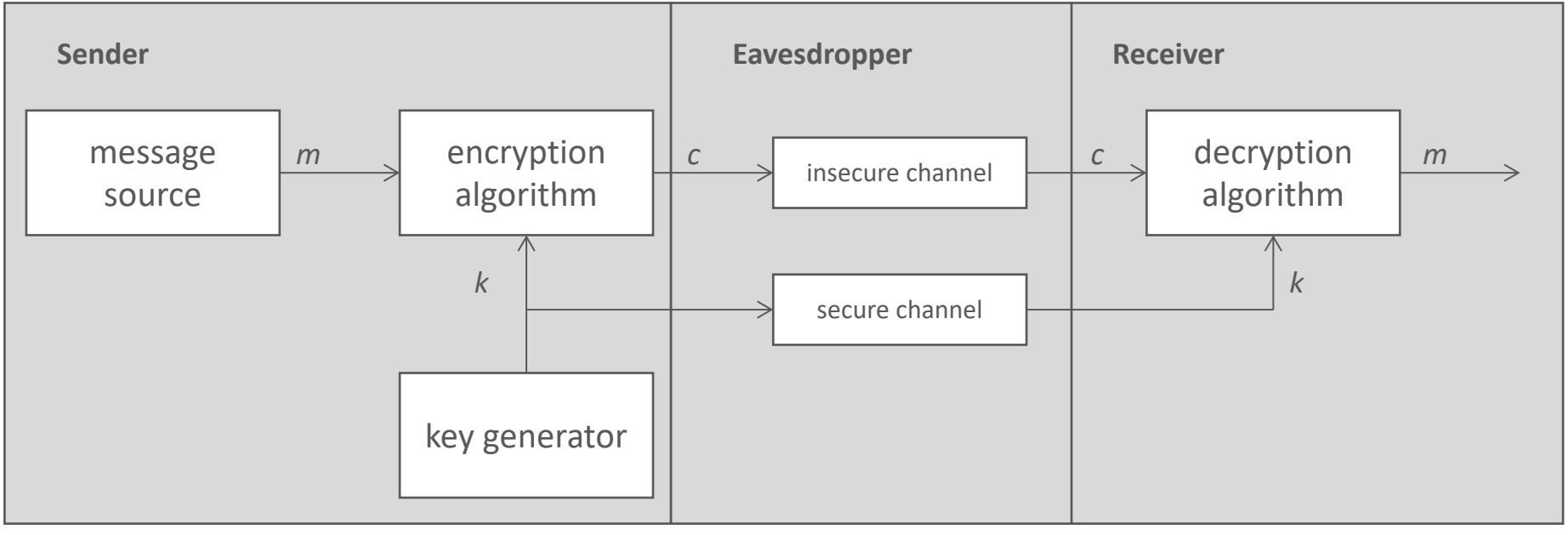
in the way that

$$D(E(M, K_E), f(K_E)) = M \quad \forall M \in \mathcal{M}, K_E \in \mathbb{D}(f)$$

A cryptographic cipher is called symmetric if $f = id$

Fundamentals of encryption

Diagram of a private-key encryption



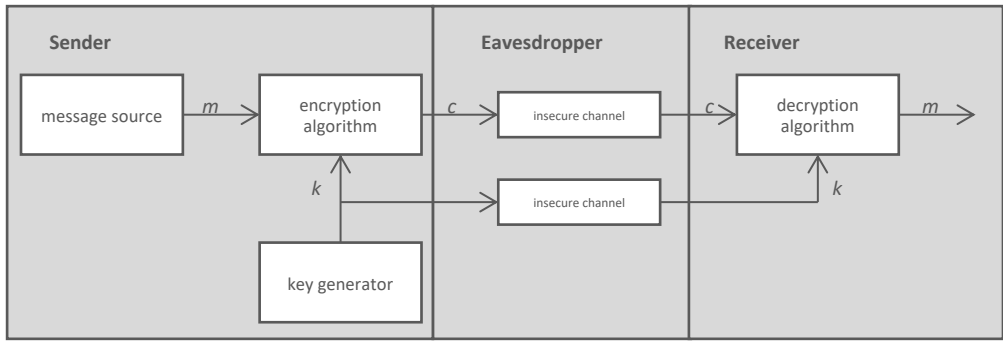
Fundamentals of encryption

Requirements:

- Security is not related to the secrecy of the functions E and D
- Quality of Cipher is only dependent on the quality of the secret (Kerckhoff-Principle)

Consequences

- big keyspaces
- good key-generators (typically RNG)



Fundamentals of encryption

Famous historic ciphers:

- Caesar-Cipher (Also know as ROT)
- Skytale
- Vigenère
- Enigma

Newer ciphers:

- DES (Data Encryption Standard)
- AES (Advanced Encryption Standard, successor of DES)
- RC4, A5, IDEA

Usual classes of operation:

- Transposition (Skytale)
- Substitution (mono- and polyalphabetic), Caesar, Vigenère
- combinations (DES,AES….)

One Time Pad (OTP)

- described first in 1882 by Frank Miller for securing telegraphy
- How it works:
  - Generate random key with same length as cleartext-message
  - ciphertext = XOR(cleartext, key)
- "information-theoretically secure":
  - ciphertext provides no information about the cleartext except the maximum possible length of the message
  - proof: any cleartext can be "decrypted" from any ciphertext message by choosing the right key => only knowledge of the "real" key delivers the "real" cleartext from the ciphertext.
- Problems:
  - long perfect **randomness** of key
  - key distribution (if I can distribute a 1MB key secretly, I can distribute a 1MB message as well without encryption)
  - using the OTP more than once makes it exploitable (that why we call it ONE Time Pad)

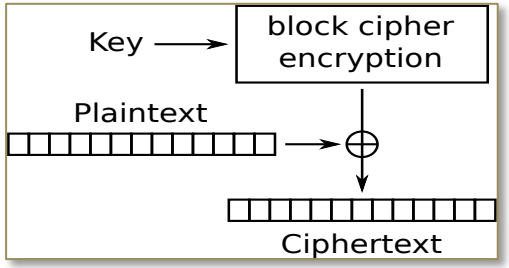
Fundamentals of encryption

Now we have a good way of getting a random key ;)

Choosing type and mode of cipher!

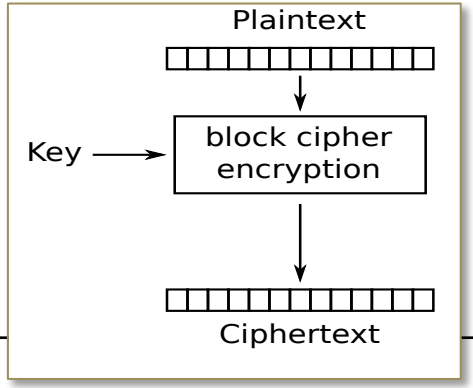
Types:

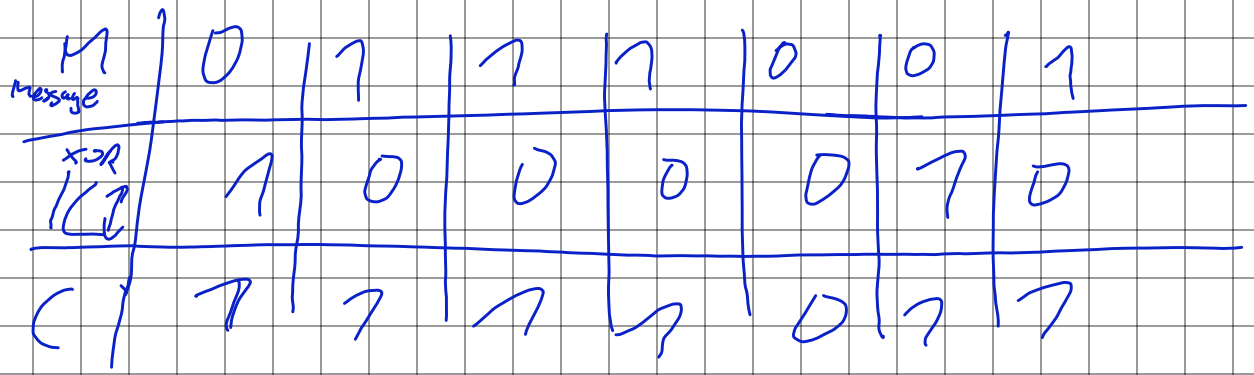
- Stream ciphers
  These work by generating a stream of pseudo-random data, then using XOR to combine the stream with the plaintext.



both types still used today

- Block ciphers
  These work by encrypting a fixed-size chunk of data (a block). Data that isn't aligned to the size of the block needs to be padded somehow. The same input always produces the same output.





Proof that the only connection between message and cipher is the key

-> if you remove key and use a new cipher but the same message, you get a new key via XOR -> mathematical proof

Domain: key length -> 5 GB of data require 5 GB key

Fundamentals of encryption

Choosing algorithm: List of popular and free algorithms:

Cipher	Key size	Type	Notes
AES	-256	block	replaced 3DES as cipher of choice
3DES	192 (112)	block	former standard still lots of implementations
Serpent	-256	block	perceived to be even more secure than AES but slower
Blowfish	-256	block	still popular, but no obvious advantage over AES or Serpent
SNOW 2.0	-256	stream	ISO/IEC standard IS 18033-4
MUGI	128	stream	ISO/IEC standard IS 18033-4
RC4	-256	stream	widely used, however take care and follow advises on not using first 12 rounds on output

If there is no reason not to do otherwise: CHOOSE AES!

ECRYPT II

http://www.ecrypt.eu.org/

Fundamentals of encryption

Now we have a good way of getting the key and a good encryption algorithm.

What next?

How long should the key be?

Key size	Time to Crack
56	399s
128	~10^{18} years
192	~$2x10^{37}$ years
256	~$3x10^{56}$ years

Age of universe: ~$14x10^{9}$ years!!!

⇨ If there is no analytical weakness in the used algorithm, 128bit will be enough for a long time, if there is an analytical weakness millions of bit keylength might be insecure (exceptions may come from weakness, which reduce the size of the key space)

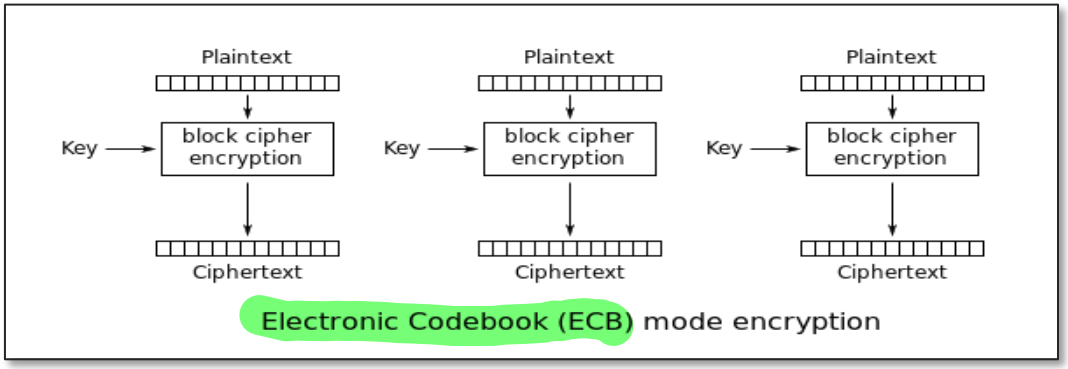
Problem: If the RNG has weaknesses a longer key might have more entropy and deliver more security… a if you don't trust your RNG fully, use longest possible keys which are fast enough!

Fundamentals of encryption

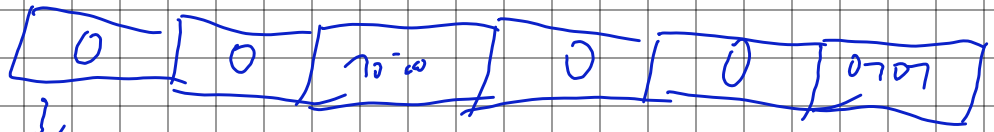
Now we have a good way of getting the key and a good encryption algorithm and the right key length! Choose cipher mode!

Popular modes:

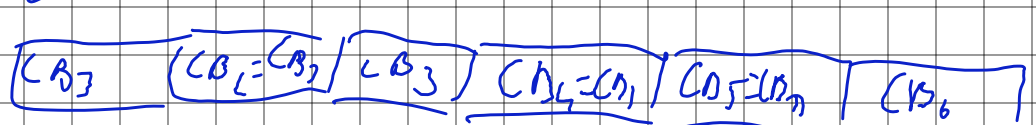
mode	description	integrity	main advantages	main disadvantages
ECB	break message into blocks and encrypt each block with raw operation	no	parallel encryption of multiblock messages	• blocks can be removed • potentially vulnerable to dictionary attacks



ECB:



ECB 2



e.g. if picture is encrypted you can see structures/outlines

→ can learn patterns and build a „dictionary"

→ dictionary attack

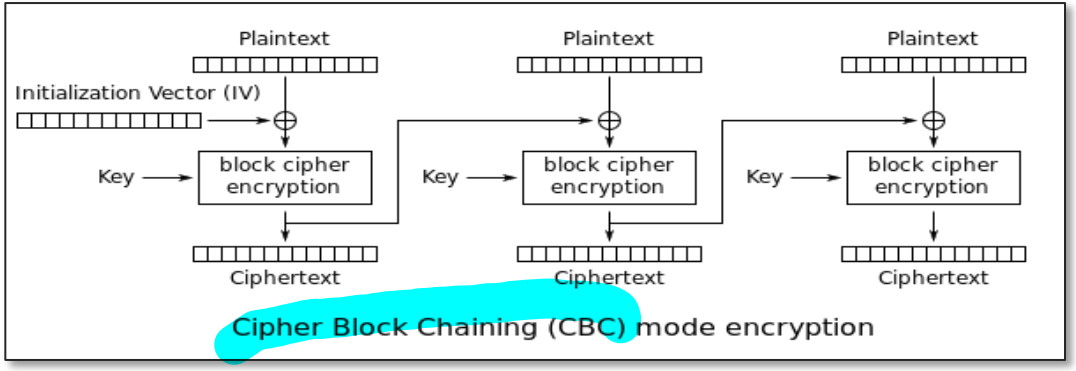
main use of ECB today is troubleshooting

Fundamentals of encryption

Popular modes:

mode	description	integrity	main advantages	main disadvantages
CBC	extension of ECB: use XOR of plaintext of current block with ciphertext of previous block, first block is XORed with initialization vector (IV)	no	• much more security than ECB • standardized by NIST • recommended by Ferguson & Schneier	• no parallel encryption • still potential dictionary attacks when used without IV

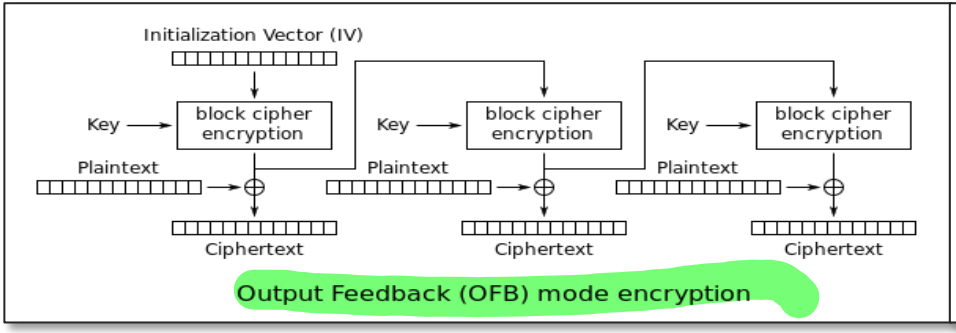
→ no more correlation in the encrypted message

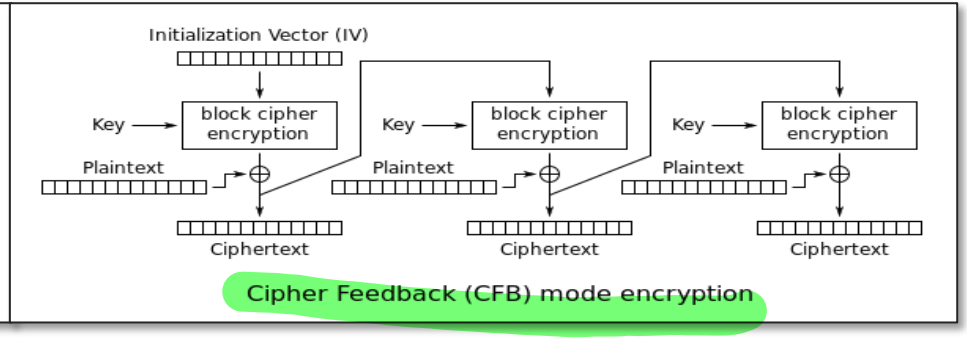


Fundamentals of encryption

Popular modes:

mode	description	integrity	main advantages	main disadvantages
OFB	streaming mode, stream is created by continually encrypting the last block of keystream to produce the next block. The first block is generated by encrypting a Nonce.	no	• keystreams can be precomputed	• bit flipping attacks are easy (generic problem of streaming modes) • reusing a key, nonce pair is critical • no parallel encryption and decryption
CFB	like OFB, but encrypting last block of ciphertext instead of keystream	no	like OFB but parallel decryption possible	

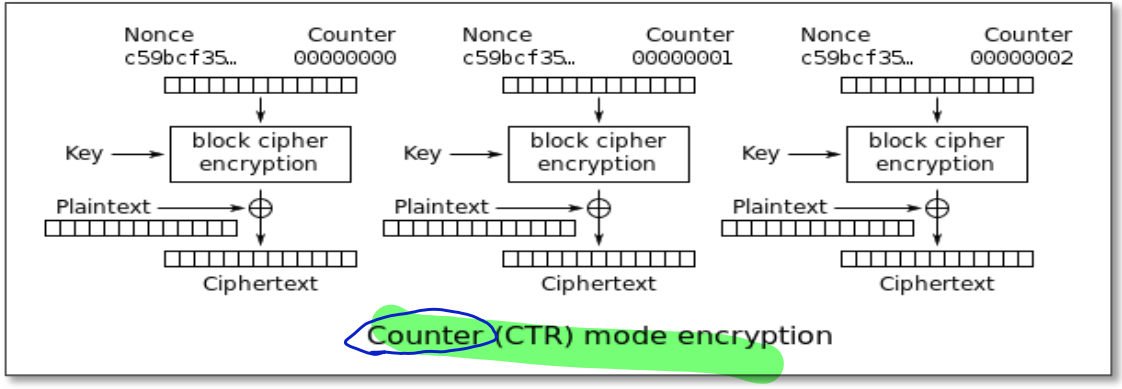


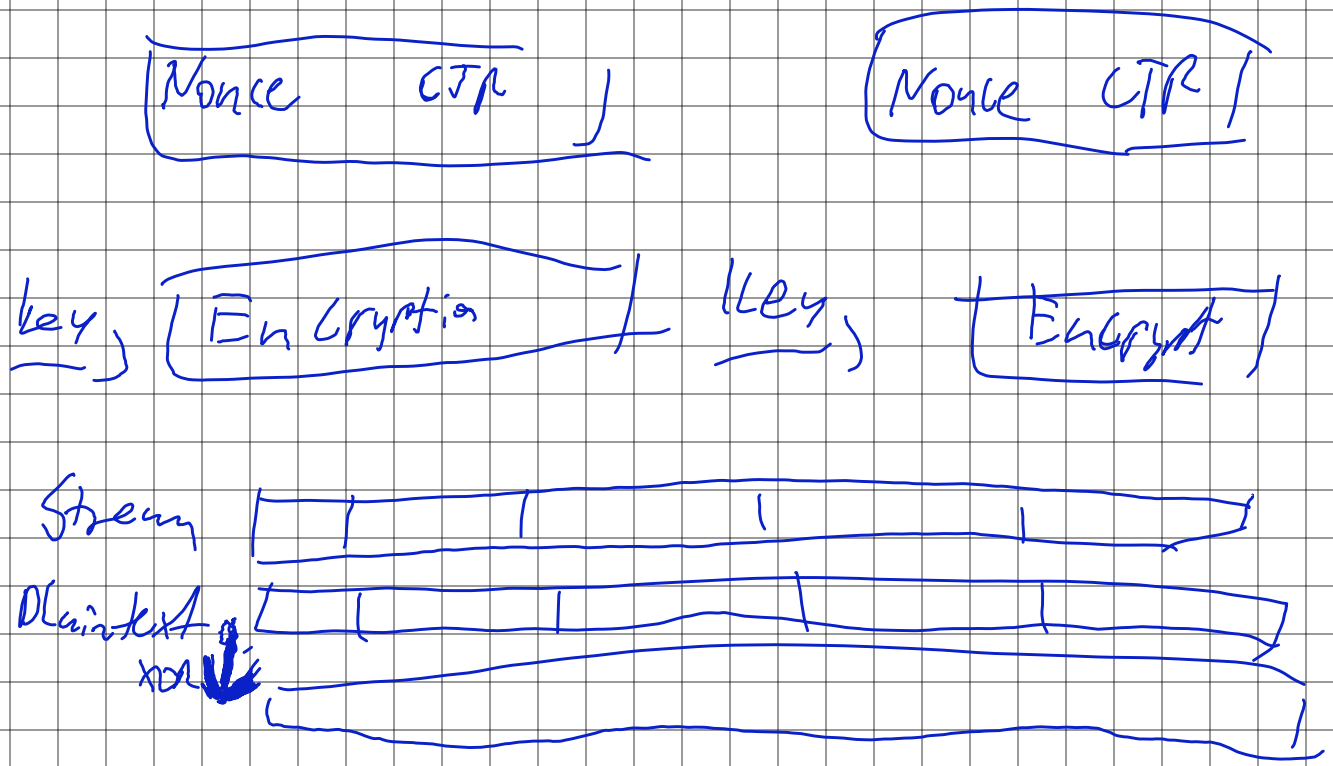


Fundamentals of encryption

Popular modes:

mode	description	integrity	main advantages	main disadvantages
CTR	streaming mode NIST description	no	• standardized by NIST • parallel encryption and decryption • recommended by Ferguson & Schneier	• IV must never be reused • length of message must be known before start

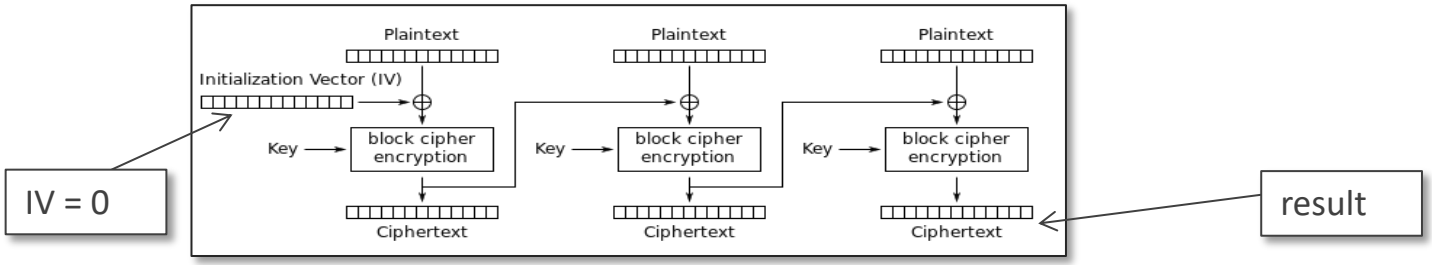



Nonce CTR
Nonce CTR
key
Encryption
key
Encrypt
Stream
Plaintext
xor

Fundamentals of encryption

Popular modes:

mode	description	integrity	main advantages	main disadvantages
CCM (CBC-MAC)	streaming mode RFC 3610 and NIST description	**yes**	• NIST recommendation • refereed security properties with single key	• IV must never be reused • length of message must be known before start

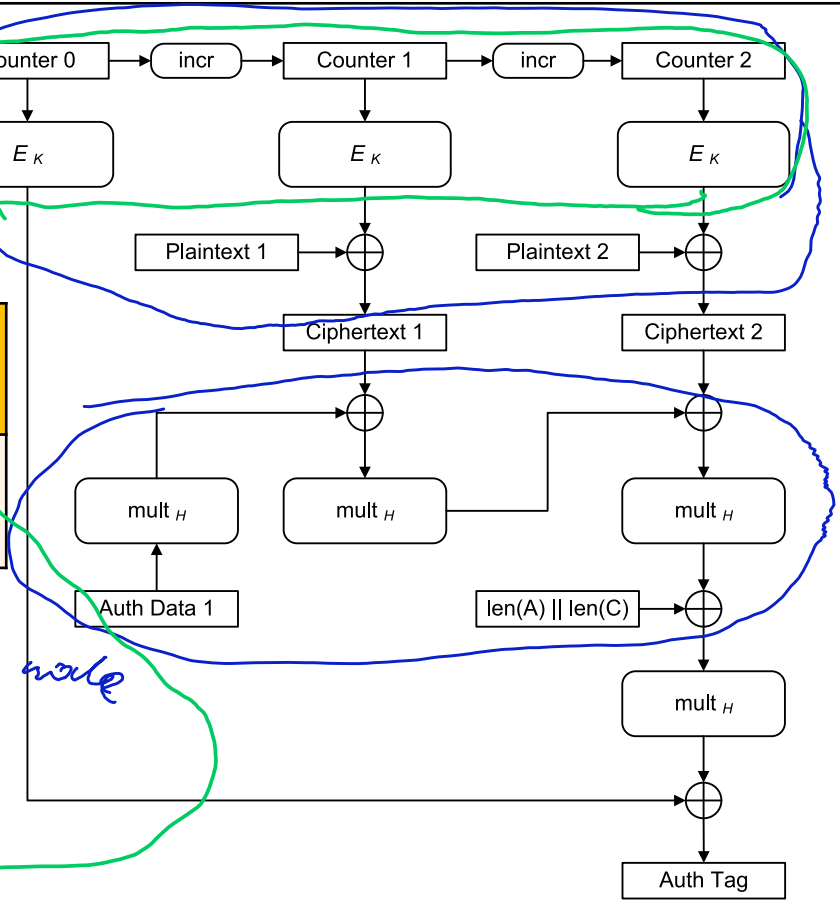


- Idea: calculate CBC and second IV zero and use the result of the final encryption as MAC

Fundamentals of encryption

Current mode of choice:

mode	description	integrity	main advantages	main disad-vantages
GCM	RFC 4106 and NIST description	yes	• NIST recommendation • refereed security	



Fundamentals of encryption

Terms used and not explained so far:

- Nonce
  - bits of data often input to cryptographic protocols and algorithms
  - only be used a single time with any particular cryptographic key
  - Sequential and random Nonces possible
  - example OFB mode
- Initialization vector (IV)
  - Nonce with additional requirements
  - must be selected in a non-predictable way a must not be sequential
  - example CBC mode

Fundamentals of encryption

Hey Ho, lets go: Functions needed from openssl to start up encrypting and decrypting:

1. Initialization

envelope

```
void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);
```

initializes cipher contex ctx

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       ENGINE *impl, unsigned char *key, unsigned char *iv);
```

- sets up cipher context ctx for encryption with cipher type from ENGINE impl.
- ctx must be initialized before calling this function.
- type is normally supplied by a function such as EVP_des_cbc().
- If impl is NULL then the default implementation is used.
- key is the symmetric key to use and
- iv is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher.
- It is possible to set all parameters to NULL except type in an initial call and supply the remaining parameters in subsequent calls, all of which have type set to NULL. This is done when the default cipher parameters are not appropriate.

Fundamentals of encryption

2. Encryption, Decryption

```
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                      int *outl, unsigned char *in, int inl);
```

- encrypts inl bytes from the buffer in and writes the encrypted version to out.
- This function can be called multiple times to encrypt successive blocks of data.
- The amount of data written depends on the block alignment of the encrypted data: as a result the amount of data written may be anything from zero bytes to (inl + cipher_block_size - 1) so outl should contain sufficient room.
- The actual number of bytes written is placed in outl.

```
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out,
                        int *outl);
```

- encrypts the « final » data, that is any data that remains in a partial block.
- It uses standard block padding (aka PKCS padding).
- The encrypted final data is written to out which should have sufficient space for one cipher block.
- The number of bytes written is placed in outl.
- After this function is called the encryption operation is finished and no further calls to EVP_EncryptUpdate() should be made.

`EVP_DecryptInit_ex(), EVP_DecryptUpdate() and EVP_DecryptFinal_ex()` are the corresponding decryption operations.

Fundamentals of encryption

side channel attacks: power consumption of key with only 0s vs with only 1s → reverse engineer the key

What to do if key shall be generated from a password?

```
int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md,
                   const unsigned char *salt,
                   const unsigned char *data, int datal, int count,
                   unsigned char *key,unsigned char *iv);
```

- derives a key and IV from various parameters.
- type is the cipher to derive the key and IV for.
- md is the message digest to use.
- The salt paramter is used as a salt in the derivation: it should point to an 8 byte buffer or NULL if no salt is used.
- data is a buffer containing datal bytes which is used to derive the keying data.
- count is the iteration count to use.
- The derived key and IV will be written to key and iv respectively.

Terms used and not explained so far:

- Salt:
  random data that helps protect against dictionary and other precomputation attacks. Generally, salt is used in password-based systems and is concatenated to the front of a password before processing. Password systems often use a one-way hash function to turn a password into an "authenticator."

Fundamentals of encryption

Good to know:

- adjust configurable parameters in ciphers with

```
int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr);
```

- query current cipher properties with

```
EVP_CIPHER *EVP_CIPHER_CTX_cipher(EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_block_size(EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_key_length(EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_iv_length(EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_mode(EVP_CIPHER_CTX *ctx);
int pad = (ctx->flags & EVP_CIPH_NO_PADDING);
int encr = (ctx->encrypt);
char *iv = (ctx->oiv);
EVP_CIPHER_CTX_mode( )
```

Fundamentals of encryption

If you are programming for Windows only: Use the Microsoft Crypto API

- Initialization

```
CryptAcquireContext(&hProvider, 0, MS_ENHANCED_PROV, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT)
```

- Key generation

```
CryptGenKey( )
CryptDeriveKey( )
CryptImportKey( )
```

- Enryption, Decryption

```
BOOL WINAPI CryptEncrypt(
  _In_     HCRYPTKEY hKey,
  _In_     HCRYPTHASH hHash,
  _In_     BOOL Final,
  _In_     DWORD dwFlags,
  _Inout_  BYTE *pbData,
  _Inout_  DWORD *pdwDataLen,
  _In_     DWORD dwBufLen
);
```

```
BOOL WINAPI CryptDecrypt(
  _In_     HCRYPTKEY hKey,
  _In_     HCRYPTHASH hHash,
  _In_     BOOL Final,
  _In_     DWORD dwFlags,
  _Inout_  BYTE *pbData,
  _Inout_  DWORD *pdwDataLen
);
```

Fundamentals of encryption

Encryption is fine, but integrity is not built in…. so how to add it

First step:

- *Cryptographic hash functions*
  - take an input string and produce a fixed-size output string (often called a hash value or message digest).
  - Given the output string, there should be no way to determine the input string other than guessing (a dictionary attack).
  - Traditional algorithms include SHA1 and MD5
  - Idea: add a hash of the message to the communication, so that the receiver can proof that the message was not changed
  - How to proof: Receiver decrypts the message and creates the hash, if it matches the transmitted hash from the sender, then the message was not changed.

Fundamentals of encryption

Important Requirements for hash-functions:

- *One-wayness*
  If given an arbitrary hash value, it should be computationally infeasible to find a plaintext value that generated that hash value.

- *Noncorrelation*
  It should also be computationally infeasible to find out anything about the original plaintext value; the input bits and output bits should not be correlated.

- Weak collision resistance
  If given a plaintext value and the corresponding hash value, it should be computationally infeasible to find a second plaintext value that gives the same hash value.

- *Strong collision resistance*
  It should be computationally infeasible to find two arbitrary inputs that give the same hash value.

- *Partial collision resistance*
  It should be computationally infeasible to find two arbitrary inputs that give two hashes that differ only by a few bits. The difficulty of finding partial collisions of size n should, in the worst case, be about as difficult as brute-forcing a symmetric key of length n/2.

Fundamentals of encryption

Choose right algorithm

Algorithm	Digest size	Security confidence	Small message speed (64 bytes), in cycles per byte[2]	Large message speed (8K), in cycles per byte
MD2	128 bits	Medium	392 cpb	184 cpb
MD4	128 bits	Insecure	32 cpb	5.8 cpb
MD5	128 bits	Very low, may be insecure	40.9 cpb	7.7 cpb
RIPEMD-160	160 bits	Medium	62.2 cpb	20.6 cpb
SHA1	160 bits	Medium, may be insecure	53 cpb	15.9 cpb
SHA-256	256 bits	Very high	119 cpb	116 cpb
SHA-384	384 bits	Very high	171 cpb	166 cpb
SHA-512	512 bits	Very high	171 cpb	166 cpb

Fundamentals of encryption

do the magic:

`EVP_MD_CTX_init()`
initializes digest context ctx.

`EVP_MD_CTX_create()`
allocates, initializes and returns a digest context.

`EVP_DigestInit_ex()`
sets up digest context ctx to use a digest type from ENGINE impl. ctx must be initialized before calling this function. type will typically be supplied by a functionsuch as EVP_sha1(). If impl is NULL then the default implementation of digest type is used.

`EVP_DigestUpdate()`
hashes cnt bytes of data at d into the digest context ctx. This function can be called several times on the same ctx to hash additional data.

`EVP_DigestFinal_ex()`
retrieves the digest value from ctx and places it in md. If the s parameter is not NULL then the number of bytes of data written (i.e. the length of the digest) will be written to the integer at s, at most EVP_MAX_MD_SIZE bytes will be written. After calling EVP_DigestFinal_ex() no additional calls to EVP_DigestUpdate() can be made, but EVP_DigestInit_ex() can be called to initialize a new digest operation.

Fundamentals of encryption

example:

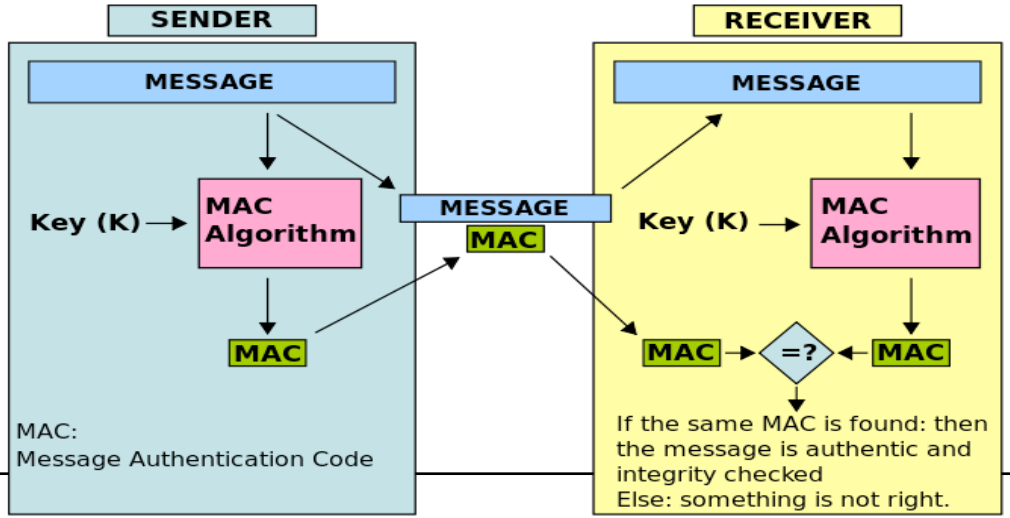
```
unsigned char *digest_message(const EVP_MD *type, unsigned char *in,
                              unsigned long n, unsigned int *outlen) {
    EVP_MD_CTX    ctx;
    unsigned char *ret;

    EVP_DigestInit(&ctx, type);
    EVP_DigestUpdate(&ctx, in, n);
    ret = (unsigned char *)malloc(EVP_MD_CTX_size(&ctx));
    EVP_DigestFinal(&ctx, ret, outlen);
    return ret;
}

int main(int argc, char *argv[  ]) {
    int           i;
    unsigned int  ol;
    if (!argv[1]) {
        printf("\nNothing to do\n");
        return 0;
    }
    unsigned char *s = argv[1];
    unsigned char *r;

    r = digest_message(EVP_sha512(  ), s, strlen(s), &ol);

    printf("SHA512(\"%s\") = ", s);
    for (i = 0;  i < ol;  i++) printf("%02x", r[i]);
    printf("\n");

    free(r);
    return 0;
}
```

Fundamentals of encryption

How to use hashes to add integrity and authentication to encryption?

Second step:

- *Message authentication codes*
  - MACs are hash functions that take a message and a secret key as input, and produce an output that cannot, in practice, be forged without possessing the secret key. This output is often called a tag. There are many ways to build a secure MAC, and there are several good MACs available, including OMAC, CMAC, and HMAC.



Fundamentals of encryption

RFC 2104:

$$HMAC\ (K,m) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel m))$$

H is a cryptographic hash function,

K is a secret key padded to the right with extra zeros to the input block size of the hash function, or the hash of the original key if it's longer than that block size,

m is the message to be authenticated,

$\parallel$ denotes concatenation,

$\oplus$ denotes exclusive or (XOR),

opad is the outer padding (0x5c5c5c…5c5c, one-block-long hexadecimal constant), and

ipad is the inner padding (0x363636…3636, one-block-long hexadecimal constant).

http://quickhash.com/hmac

Fundamentals of encryption

do the magic with HMAC (widely used: RFC2104):

```
void spc_incremental_hmac(unsigned char *key, size_t keylen) {
    int           i;
    HMAC_CTX      ctx;
    unsigned int  len;
    unsigned char out[20];

    HMAC_Init(&ctx, key, keylen, EVP_sha1(  ));
    HMAC_Update(&ctx, "fred", 4);
    HMAC_Final(&ctx, out, &len);
    for (i = 0;  i < len;  i++) printf("%02x", out[i]);
    printf("\n");

    HMAC_Init(&ctx, 0, 0, 0);
    HMAC_Update(&ctx, "fred", 4);
    HMAC_Final(&ctx, out, &len);
    for (i = 0;  i < len;  i++) printf("%02x", out[i]);
    printf("\n");
    HMAC_cleanup(&ctx); /* Remove key from memory */
}
```

thanks for your interest

to be continued