C R Y

Table of Contents

Acronyms and Keywords....5
Introduction....6
  Implemented primitives....6
  Licensing....8
  Source Code....8
Symmetric Block Ciphers....9
  DES....9
    Technical Overview....9
    Triple DES....10
    API....10
  AES....12
    Technical Overview....12
    API....13
  Modes of operation....15
    Electronic Code Book (ECB)....15
    Cipher Block Chaining (CBC)....16
    Cipher Feedback (CFB)....16
    Output Feedback (OFB)....17
    Counter mode (CTR)....18
    Galois Counter mode (GCM)....18
    API....19
    Experiment - ECB Information Leakage....23
Stream Ciphers....24
  Trivium....24
    Technical Overview....24
    API....25
  ARC4....27
    Technical Overview....27
    API....28
Public Key Algorithms....30
  RSA....31
    Technical Details....31
    API....33
  Diffie-Hellman....34
    Technical Details....34
    API....35
  Elgamal Digital Signature....36
    Technical Details....37
    API....37
  DSA....39
    Technical Details....39

API....40
Hash Functions....42
MD5....43
Technical Details....43
API....44
SHA-1....45
Technical Details....45
API....46

Acronyms and Keywords

API	Application Programming Interface
NIST	(US) National Institute for Standards and Technology
NSA	(US) National Security Agency
FIPS	(US) Federal Information Processing Standard
DES	Data Encryption Standard cryptosystem
AES	Advanced Encryption Standard cryptosystem
IV	Initialization Vector
PK	Pyblic Key cryptography
RSA	Rivest Shamir Adleman cryptosystem
DSA	Digital Signature Algorithm
MD	Message Digest
SHA	Secure Hash Algorithm
PRNG	Pseudo Random Number Generator
CSPRNG	Cryptographicaly Secure PRNG
MPI	Multiple Precision Integer

Introduction

CRY provides a simple and portable implementation of a good set of security related primitives with focus on cryptography.

This documentation contains the CRY API reference complemented with a short and intuitive glimpse of the theory behind each presented primitive.

This is not an introduction to cryptography

You may find that some information about the algorithms implementation, mathematical background and proof of correctness may be scattered and quite incomplete. Again, only a glimpse to the background theory is given, for more information the interested reader shall refer to one of the several good books about cryptography.

This document is a continuous *work in progress* and some of the algorithms included in the CRY library may not have be covered yet.

For missing API information (e.g. detailed explanation of the functions parameters) please refer to the *Doxygen* documentation available here: https://crylib.gitlab.io/cry.

Implemented primitives

Follows an exaustive list of the primitives included in the CRY library up to the last version.

Symmetric ciphers

Block ciphers

- AES : advanced encryption standard (Rijndael)
- DES and Triple DES : data encryption standard

Block ciphers modes of operation

- ECB : electronic codebook
- CBC : cipher block chain
- CFB : cipher feedback
- CTR : counter mode
- GCM : Galois counter mode

Stream ciphers

- ARC4
- Trivium

Public key algorithms

Ciphers

- RSA (PKCS1 v1.5 padding)

Secret exchange

- DH : Diffie-Hellman
- ECDH : Elliptic Curve Diffie-Hellman

Digital signature

- RSA (PKCS1 v1.5 padding)
- DSA
- ECDSA
- Elgamal

Pseudo random number generators

- AES-CTR CSPRNG
- LFSR-113 PRNG

Message Authentication Code

- HMAC
- CMAC

Secure Hash

- MD5
- SHA1
- SHA256

Multiple precision integers

- Basic arithmentic (add,sub,mul,div,mod,abs,exp,sqr)
- Modular exponentiation
- Modular inverse (Euclidean)
- GCD and LCM
- Probabilistic prime numbers generator (Miller-Rabin)
- Random mpi generator
- Comba, Karatsuba and Toom-Cook-3 multipliers

Elliptic curves

- EC group arithmetic.
- NIST recommended elliptic curve domain parameters over Fp.
- Brainpool standard curve domain parameters over Fp (RFC 5639).

Cyclic redundancy checks

- CRC16-CCITT
- CRC16-IBM
- CRC16-DNP
- CRC32-Ethernet

Classical ciphers

- Hill cipher
- Polyalphabetic Affine cipher

Licensing

CRY is **Free and Open Source Software** licensed under the permissive **MIT License**.

MIT License grants the software end user rights such as copying, modifying, merging, distributing, etc.

MIT License permits reuse within **proprietary** software provided that all copies of the licensed software include a copy of the MIT license terms and the copyright notice.

MIT License is also compatible with many **copyleft** licenses, such as the GPL: MIT licensed software can be integrated into GPL software, but not the other way around.

```
MIT License

Copyright (c) 2013-2019 The CRY Authors

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Source Code

- **Gitlab**: https://gitlab.com/crylib/cry
- **Github**: https://github.com/crylib/cry

Symmetric Block Ciphers

The block cipher is an algorithm operating on fixed-length groups of bits, called **blocks**, with a transformation that is specified by a symmetric key.

All modern strong block ciphers are built over two strong encryption core primitives:

- **Confusion**: operation where the relationship between key and ciphertext is obscured.
- **Diffusion**: operation where the influence of one plaintext symbol is spread over many ciphertext symbols with the goal of hiding statistical properties of the plaintext.

DES

Data Encryption Standard block cipher was developed by **IBM** and refined **NSA** in the early **1970**.

Has been one of the most analyzed ciphers in the history and has been a wordwide standard for more than twenty years (now replaced by AES).

The intense academic scrutiny the algorithm received over time led to the modern understanding of block ciphers and their cryptanalysis.

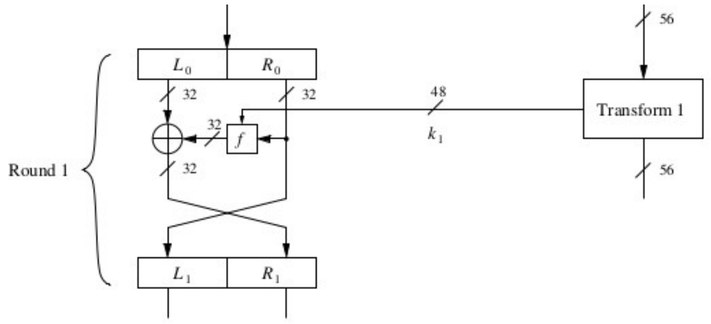
The algorithm process **blocks** of **64 bits** with a **key** of **56 bits** and it is based on a **Feistel Network** structure.

Two major security related criticisms against DES cryptosystem exists:

1. **Key space** is too small, i.e. the algorithm is vulnerable against brute-force attacks.
2. **Design criteria** of some components were kept secret, thus there were always the suspect that NSA modified the algorithm to insert a trapdoor for its own usage.

Technical Overview

For each block of plaintext, encryption is handled in **16 rounds** which all perform the identical operation. In every round a different 48-bit subkey k_i, derived from the main key, is used.



The “*Transform*” step is used to derive the current round sub-key and is known as **key-schedule** function.

On the *i*-th round, after that the 64-bit input has been sliced in two 32-bit chunks, L_i and R_i, the round output is set: $L_{i+1}=R_i$ and $R_{i+1}=f(k_i, R_i)\oplus L_i$.

The ***f* function** plays a crucial role for the security of DES and is mainly composed of bit-substitution via eight lookup tables known as **S-boxes**.

S-boxes are the core of the DES in term of cryptographic strength since they are the only **nonlinear element** in the algorithm providing **confusion**.

Without a nonlinear building block, an attacker could express the DES input and output with a system of linear equations where the key bits are the unknowns.

After that each S-box substitution is applied, a bit **permutation** is performed to provide **diffusion**.

At the begin and at the end of the 16 rounds a standard defined permutation of the input block bits is performed. The purpose of this permutation is not yet clear, since it doesn’t add any security to the overall algorithm.

With a Feistel Network structure, encryption and decryption are almost the same operation; decryption only requires a reversed key schedule, i.e. we start from k_{16} instead from k_1 .

Triple DES

To overcome DES short key length issue, for each 64-bit data block, triple DES (3DES or **TDES**) applies the DES cipher three times in a row. TDES is thus able to use three DES keys each of 56 bits.

The TDES algorithm is commonly used in **EDE** mode, that is the plaintext is encrypted with the first key, is decrypted using the second key and then is finally encrypted using the third key.

$$ciphertext=E_{k3}(D_{k2}(E_{k1}(plaintext)))$$
$$plaintext\ =D_{k1}(E_{k2}(D_{k3}(plaintext)))$$

EDE mode allows backward compatibility with plain DES in case that the three keys are equal.

API

The DES and TDES algorithms are accessible via the same API. The algorithm type is inferred internally from the key length argument.

Initialization

Zero out the context.

```
void cry_des_init(cry_des_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_des_clear(cry_des_ctx *ctx);
```

Key set

Set the cipher key (8 bytes for DES, 24 for TDES).

```
void cry_des_key_set(cry_des_ctx *ctx, const unsigned char *key, size_t size);
```

Encrypt

```
void cry_des_encrypt(cry_des_ctx *ctx, unsigned char *dst,
                     const unsigned char *src, size_t size);
```

Decrypt

```
void cry_des_decrypt(cry_des_ctx *ctx, unsigned char *dst,
                     const unsigned char *src, size_t size);
```

Usage Example

```
cry_des_ctx des;                                 /* DES context */
unsigned char buf[16];                           /* Encryption/decryption buffer */
unsigned char *key = "0123456789abcdefABCDEF!!"; /* 24 bytes key */
unsigned char *msg = "HelloWorld------";         /* Message padded up to a multiple of 8 */

/* DES */
cry_des_init(&des);                   /* Context initialization */
cry_des_key_set(&des, key, 8);        /* Use the first 8 key bytes */
cry_des_encrypt(&des, buf, msg, 16);  /* Encrypt */
cry_des_decrypt(&des, buf, buf, 16);  /* Decrypt (store decrypted message in the same buffer) */
cry_des_clear(&des);                  /* Context cleanup */

/* TDES */
cry_des_init(&des);                   /* Context initialization */
cry_des_key_set(&des, key, 24);       /* Use all 24 key bytes */
cry_des_encrypt(&des, buf, msg, 16);  /* Encrypt */
```

```
    cry_des_decrypt(&des, buf, buf, 16);  /* Decrypt (store decrypted message in the same buffer) */
    cry_des_clear(&des);                  /* Context cleanup */
```

AES

Advanced Encryption Standard is a subset of the **Rijndael** block cipher developed by two belgian cryptographers **Rijmen** and **Deamen** in 1998, standardized by **NIST** in **2001** as the successor of DES.

The general Rijndael block and key size vary between 128, 192 and 256 bits. However, the AES standard only calls for a **block** size of **128 bits**.

AES is based on a design principle known as a **Substitution-Permutation Network**.

Till date, no practical cryptanalytic attacks against AES has been discovered.

According to NSA, the design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information require use of either the 192 or 256 key lengths.

Technical Overview

As for the others block ciphers AES is composed by a number of rounds performing substitution and permutation operations.

The number of rounds depends on the key length.

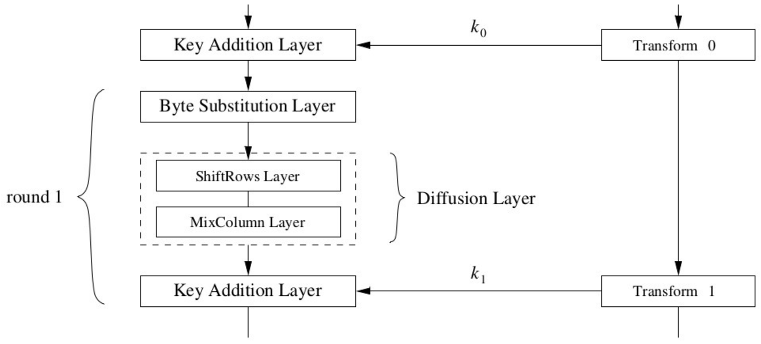
key lengths	# rounds = n_r
128 bit	10
192 bit	12
256 bit	14

In contrast to DES, AES does not have a Feistel structure and encrypts an entire block in one iteration.

Each **round**, with the exception of the first, consists of **three layers**:

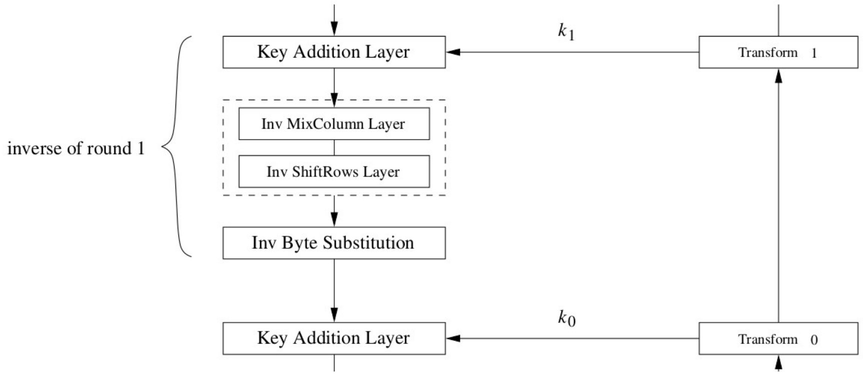
- **Substitution** (**S-box**): each element of the state is non-linearly transformed using lookup tables with special mathematical properties (confusion).
- **Diffusion** (**Shift Rows** + **Mix Column**): provides diffusion over all state bits.
- **Key addition**: the 128-bit round key, derived from the main key, is XORed to the state.

The first round only has the Key Addition layer.



AES is **byte oriented**, which means that the round operations are performed at byte level. This is in contrast to DES, which makes heavy use of bit permutation and can thus be considered to have a bit-oriented structure.

Because AES is not based on a Feistel network, for decryption all the layers shall be inverted, i.e. for each round Key Addition Layer comes before the Diffusion Layer.



The last round only has the Key Addition layer.

API

The CRY implementation supports the three standard key lengths, i.e. 128, 196 and 256. The choice is performed by the user when setting the cipher key.

Initialization

Zero out the context.

```
void cry_aes_init(cry_aes_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_aes_clear(cry_aes_ctx *ctx);
```

Key set

Set the cipher key. Valid lengths are 16, 24 and 32 in order to use AES 128, 192 and 256, respectively.

```
void cry_aes_key_set(cry_aes_ctx *ctx, const unsigned char *key, size_t size);
```

Encrypt

```
void cry_aes_encrypt(cry_aes_ctx *ctx, unsigned char *dst,
                     const unsigned char *src, size_t size);
```

Decrypt

```
void cry_aes_decrypt(cry_aes_ctx *ctx, unsigned char *dst,
                     const unsigned char *src, size_t size);
```

Usage example:

```
cry_aes_ctx aes;                           /* AES context /
unsigned char buf[16];                     /* Encryption/decryption buffer */
unsigned char *key = "000102030405060708090a0b0c0d0e0f"; /* 32 bytes key */
unsigned char *msg = "HelloWorld------";   /* Message padded up to a multiple of 16 */

/* AES-128 */
cry_aes_init(&aes);                   /* Context initialization */
cry_aes_key_set(&aes, key, 16);       /* Use the first 16 key characters */
cry_aes_encrypt(&aes, buf, msg, 16);  /* Encrypt */
cry_aes_decrypt(&aes, buf, buf, 16);  /* Decrypt (store decrypted message in the same buffer) */
cry_aes_clear(&aes);                  /* Context cleanup */

/* AES-256 */
cry_aes_init(&aes);                   /* Context initialization */
cry_aes_key_set(&aes, key, 32);       /* Use all the 32 key characters */
cry_aes_encrypt(&aes, buf, msg, 16);  /* Encrypt */
cry_aes_decrypt(&aes, buf, buf, 16);  /* Decrypt (store decrypted message in the same buffer) */
cry_aes_clear(&aes);                  /* Context cleanup */
```

Modes of operation

A mode of operation describes how to repeatedly apply a block cipher single-block operation to securely transform amounts of data larger than a block.

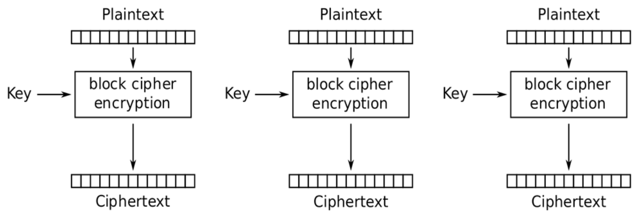
Most modes require a unique sequence, called an **initialization vector (IV)**, for each encryption operation. The IV has to be non-repeating and, for some modes, random as well. The initialization vector is used to ensure that distinct ciphertext are produced even when the same plaintext is encrypted multiple times independently with the same key.

Some block cipher modes operate on whole blocks and require that the last part of the data be padded to a full block if it is smaller than the current block size, there are other modes that do not require padding and thus they effectively transform the block cipher in a stream cipher.

Electronic Code Book (ECB)

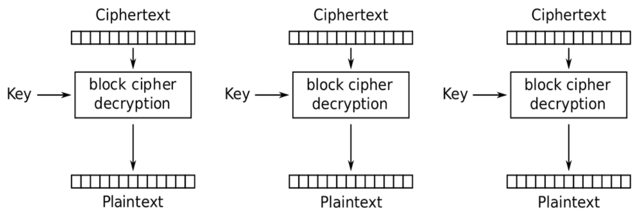
The message is divided into blocks and each block is encrypted separately.

$$C_i = E_k(P_i)$$



Electronic Codebook (ECB) mode encryption

$$P_i = D_k(C_i)$$



Electronic Codebook (ECB) mode decryption

Characteristics:

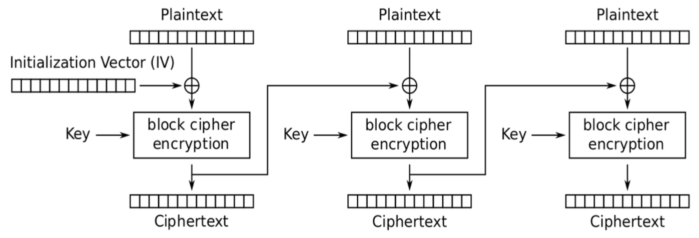
- Encryption parallelizable: yes
- Decryption parallelizable: yes
- Random read access: yes

The disadvantage of this method is a lack of diffusion. Because ECB encrypts identical plaintext blocks into identical ciphertext blocks, it does not hide data patterns well. In some senses, it doesn't provide serious message confidentiality, and it is not recommended for use in cryptographic protocols at all.

Cipher Block Chaining (CBC)

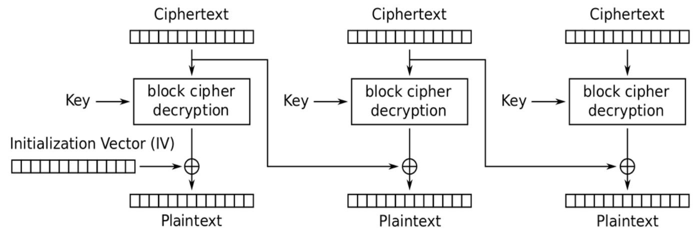
Each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an IV must be used in the first block.

$$C_0 = IV$$
$$C_i = E_k(P_i \oplus C_{i-1})$$



Cipher Block Chaining (CBC) mode encryption

$$C_0 = IV$$
$$P_i = D_k(C_i) \oplus C_{i-1}$$



Cipher Block Chaining (CBC) mode decryption

Characteristics:

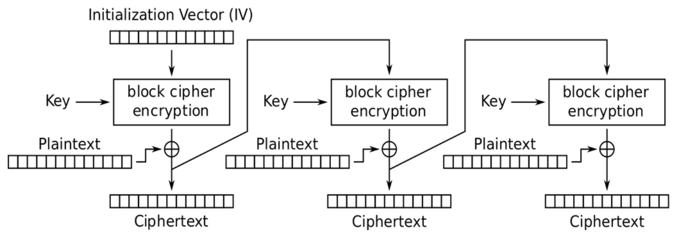
- Encryption parallelizable: no
- Decryption parallelizable: yes
- Random read access: yes

Decrypting with the incorrect IV causes the first block of plaintext to be corrupt but subsequent plaintext blocks will be correct.

Cipher Feedback (CFB)

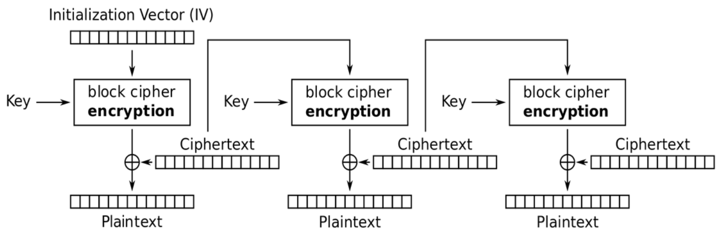
Turns a block cipher into a self-synchronizing stream cipher. Operation is very similar to CBC, in particular, CFB decryption is almost identical to CBC encryption performed in reverse.

$$C_0 = IV$$
$$C_i = E_k(C_{i-1}) \oplus P_i$$



Cipher Feedback (CFB) mode encryption

$$C_0 = IV$$
$$P_i = E_k(C_{i-1}) \oplus C_i$$



Cipher Feedback (CFB) mode decryption

Characteristics:

- Encryption parallelizable: no
- Decryption parallelizable: yes
- Random read access: yes

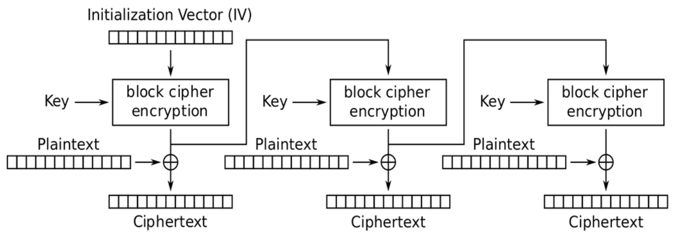
CFB shares three advantages over CBC mode with the stream cipher modes OFB, CTR and GCM:

- only the encrypt block cipher primitive is used (no decryption)
- the message does not need to be padded to a multiple of the cipher block size.
- just as other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error correcting codes to function normally even when applied before encryption.

Output Feedback (OFB)

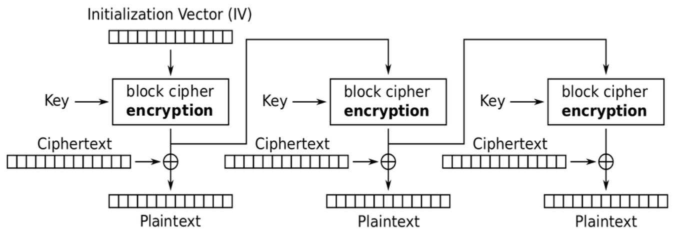
Turns a block cipher into a synchronous stream cipher with a construction similar to the CFB mode of operation. Given O_i the *i*-th block cipher output.

$$O_0 = IV$$
$$O_i = E(O_{i-1})$$
$$C_i = O_i \oplus P_i$$



Output Feedback (OFB) mode encryption

$$O_0 = IV$$
$$O_i = E(O_{i-1})$$
$$P_i = O_i \oplus C_i$$



Output Feedback (OFB) mode decryption

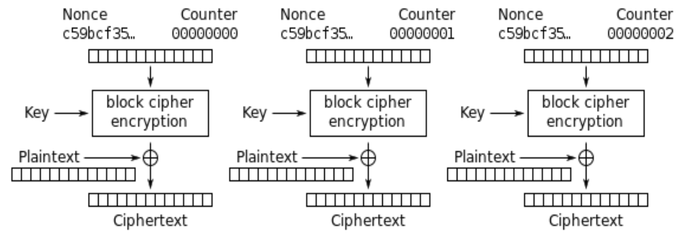
Characteristics:

- Encryption parallelizable: no
- Decryption parallelizable: no
- Random read access: no

Counter mode (CTR)

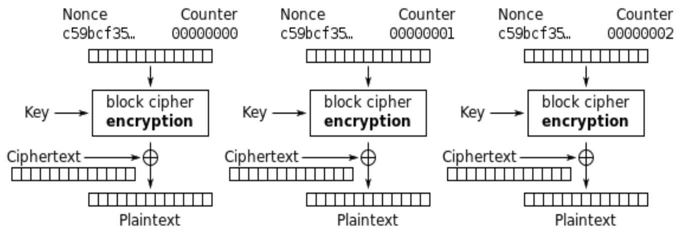
Turns a block cipher into a synchronous stream cipher. A randomly seeded counter is encrypted and then XORed with the block to encrypt/decrypt.

$$count_0 = IV$$
$$count_i = count_{i-1} + 1$$
$$C_i = E(count_i) \oplus P_i$$



Counter (CTR) mode encryption

$$count_0 = IV$$
$$count_i = count_{i-1} + 1$$
$$P_i = E(count_i) \oplus C_i$$



Counter (CTR) mode decryption

Characteristics:

- Encryption parallelizable: yes
- Decryption parallelizable: yes
- Random read access: yes

In theory, the counter can be any function which produces a sequence which is guaranteed not to repeat for a long time.

Galois Counter mode (GCM)

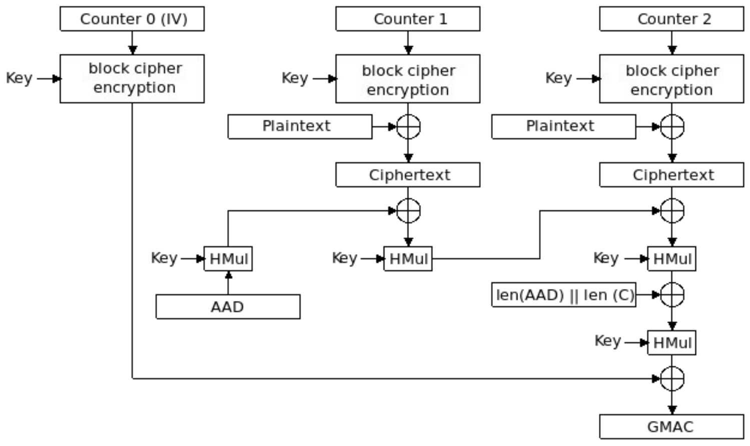
Provides both confidentiality and authentication services in a single cryptographic primitive.
The algorithm is capable to add additional authentication data (**AAD**) to the procedure to be included in the finally generated message authentication code (**GMAC**).
Modes such GCM are known as **AEAD** (Authenticated Encryption with Additional Data) ciphers.

The encryption algorithm is pretty similar to the classic CTR mode, with the exception that the counter increment is performed in the the Galois field $GF(2^{128})$.
Multiplications in the authentication procedure (HMul) are performed in $GF(2^{128})$ as well.

$$count_0 = IV$$
$$count_i = count_{i-1} + 1$$
$$C_i = E(count_i) \oplus P_i \quad , i > 0$$

$$H_0 = HMul(AAD)$$
$$H_i = HMul(H_{i-1} \oplus C_i)$$

$$L = len(AAD) \| len(Text)$$
$$K = HMul(H_{last} \oplus L)$$
$$GMAC = E(count_0) \oplus K$$



The inverse operation can be easily deduced.

Characteristics:

- Encryption parallelizable: yes
- Decryption parallelizable: yes
- Random read access: yes

API

ECB mode is provided directly by using the block cipher API (setting input size).

To avoid repetitions, CBC, CFB, OFB and CTR modes are presented by using a generic API that is "compatible" with all the four modes by replacing the specific mode name with the placeholder "xxx".

Block Cipher Context and Interface

The mode of operation context structure is used to maintain the operational state along with pointers to the backing block cipher (e.g. AES) context and interface.

A cipher interface structure is a collection of function pointers used internally by the mode of operation to perform primitive actions over the backing block cipher, such as key set and block encryption.

```
struct cry_ciph_itf {
    cry_ciph_init_f    init;
    cry_ciph_clean_f   clean;
    cry_ciph_key_set_f key_set;
    cry_ciph_encrypt_f encrypt;
    cry_ciph_decrypt_f decrypt;
};
```

This design allows to easily exploit any implementation of the block cipher implementation, for example a hardware engine.

Initialization

Zero out the context and set the backing block cipher context and interface.

```
void cry_xxx_init(cry_xxx_ctx *ctx, void *ciph_ctx, const cry_ciph_itf *ciph_itf);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_xxx_clear(cry_xxx_ctx *ctx);
```

Key set

Set the cipher key. The key size shall be equal to the key size of the backing cipher.

```
void cry_xxx_key_set(cry_xxx_ctx *ctx, const unsigned char *key, size_t size);
```

IV set

Set the Initialization Vector. Setting a new initialization vector resets the cipher, the key is maintained. The IV length is typically equal to the cipher block length (except GCM), shorter IV are zero padded.

```
void cry_xxx_iv_set(cry_xxx_ctx *ctx, const unsigned char *iv, size_t size);
```

Encrypt

```
void cry_xxx_encrypt(cry_xxx_ctx *ctx, unsigned char *dst,
                     const unsigned char *src, size_t size);
```

Decrypt

```
void cry_xxx_decrypt(cry_aes_ctx *ctx, unsigned char *dst,
                     const unsigned char *src, size_t size);
```

GCM mode adds to the general interface, the functions to *update* and *digest*.
When used with a cipher with a 128-bit block (e.g. AES-GCM), if the IV is 96-bit (12 bytes) then this is not zero padded and is used to implement a slightly optimized GCM mode defined by the standard.

Update

Set (optional) Additional Authentication Data. This function shall be called before encrypt function.

```
void cry_gcm_update(struct cry_gcm_ctx *ctx, const unsigned char *aad, size_t size);
```

Digest

Get the message digest (GMAC).

```
void cry_gcm_digest(struct cry_gcm_ctx *ctx, unsigned char *mac, size_t size);
```

Usage Example : AES-128-CBC

```
cry_cbc_ctx cbc;                              /* CBC context */
cry_aes_ctx aes;                              /* AES context */
cry_ciph_itf aes_itf;                         /* AES interface */
unsigned char *key = "0123456789abcdef";      /* AES 128 bit key */
unsigned char *iv  = "oaisdfjoajksafkl";      /* Initialization Vector */
unsigned char *msg = "HelloWorld------";      /* Message padded up to a multiple of 16 */
unsigned char buf[16];                        /* Encryption/decryption buffer */

/* Initialize AES interface */
aes_itf.key_set = cry_aes_key_set;            /* AES key-set callback */
aes_itf.encrypt = cry_aes_encrypt;            /* AES encrypt callback */
aes_itf.decrypt = cry_aes_decrypt;            /* AES decrypt callback */

/* Initialize */
cry_cbc_init(&cbc, &aes, &aes_itf);           /* CBC context initialization using AES-128 */
cry_cbc_key_set(&cbc, key, 16);               /* Key set */

/* Encrypt */
cry_cbc_iv_set(&cbc, iv, 16);                 /* IV set */
cry_cbc_encrypt(&cbc, buf, msg, 16);          /* Encrypt */

/* Decrypt */
cry_cbc_iv_set(&cbc, iv, 16);                 /* IV set */
cry_cbc_decrypt(&cbc, buf, buf, 16);          /* Decrypt (store output in input buffer) */
```

As with ECB, the processed data length shall be a multiple of the backing cipher block size.

Usage Example : DES-CTR

```
cry_ctr_ctx ctr;                              /* CTR context */
cry_des_ctx des;                              /* DES context */
cry_ciph_itf des_itf;                         /* DES interface */
unsigned char *key = "01234567";              /* DES key */
unsigned char *iv  = "oaisdfjo";              /* Initialization Vector */
unsigned char *msg = "HelloWorld";            /* Arbitrary message */
unsigned char buf[10];                        /* Encryption/decryption buffer */

/* Initialize DES interface (decrypt not required) */
```

```
des_itf.key_set = cry_des_key_set;              /* DES key-set callback */
des_itf.encrypt = cry_des_encrypt;              /* DES encrypt callback */

/* Initialize */
cry_ctr_init(&ctr, &des, &des_itf);             /* CTR context initialization using DES */
cry_ctr_key_set(&ctr, key, 8);                  /* Key set */

/* Encrypt */
cry_ctr_iv_set(&ctr, iv, 8);                    /* IV set */
cry_ctr_encrypt(&ctr, buf, msg, 10);            /* Encrypt */

/* Decryption */
cry_ctr_iv_set(&ctr, iv, 8);                    /* IV set */
cry_ctr_decrypt(&ctr, buf, buf, 10);            /* Decrypt (store output in input buffer) */
```

As a stream cipher, the processed data length may not be a multiple of the backing cipher block size.

Usage Example : AES-256-GCM

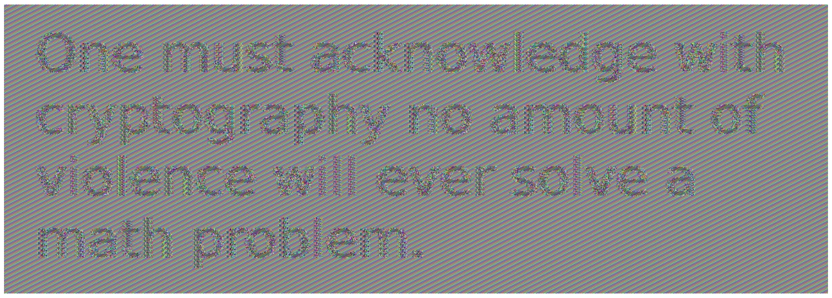
```
cry_gcm_ctx gcm;                                /* GCM context */
cry_aes_ctx aes;                                /* AES context */
cry_ciph_itf aes_itf;                           /* AES interface */
unsigned char *key = "0102030405060708111213141516l718";   /* AES 256 bit key */
unsigned char *iv = "oaisdfjoajks";             /* 12 bytes Initialization Vector */
unsigned char *aad = "myaad”;                   /* Additional authentication data */
unsigned char *msg = "HelloWorld";              /* Arbitrary message */
unsigned char mac1[16], mac2[16];               /* GMAC buffers */
unsigned char buf[10];                          /* Encrypt/decrypt buffer */

/* Initialize AES interface (decrypt not required) */
aes_itf.key_set = cry_aes_key_set;              /* AES key-set callback */
aes_itf.encrypt = cry_aes_encrypt;              /* AES encrypt callback */

/* Initialize GCM to use AES-256 */
cry_gcm_init(&gcm, &aes, &aes_itf);             /* GCM context initialization using AES-256 */
cry_gcm_key_set(&gcm, key, 32);                 /* Key set */

/* Encrypt and GMAC generation */
cry_gcm_iv_set(&gcm, iv, 12);                   /* IV set */
cry_gcm_update(&gcm, aad, 5);                   /* AAD set */
cry_gcm_encrypt(&gcm, buf, msg, 10);            /* Encrypt */
cry_gcm_digest(&gcm, mac1, 16);                 /* Digest (GMAC get) */

/* Decrypt and MAC verification */
cry_gcm_iv_set(&gcm, iv, 16);                   /* IV set */
cry_gcm_update(&gcm, aad, 5);                   /* AAD set */
cry_gcm_decrypt(&gcm, buf, buf, 10);            /* Decrypt (store output in input buffer) */
cry_gcm_digest(&gcm, mac1, 16);                 /* Digest (GMAC get) */

check_ok = (memcmp(mac1, mac2, 16) == 0);       /* GMAC check */
```

As a stream cipher, the processed data length may not be a multiple of the block cipher block size.
A message may be just encrypted, just authenticated or both. To just authenticate some data the “encrypt/decrypt” API functions shall be skipped; to only encrypt a message the “update” API function shall be skipped.

Experiment - ECB Information Leakage

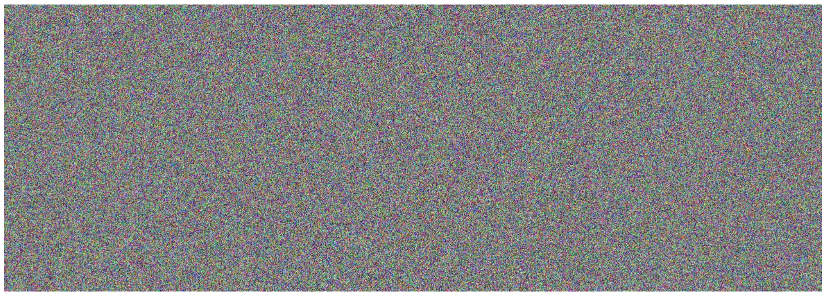
This example practically shows why using a block ciphers in ECB mode is not a good idea to properly guarantee information confidentiality.

The following picture is the AES-ECB encrypted version of an image containing some text.


One must acknowledge with
cryptography no amount of
violence will ever solve a
math problem.

Because of cleartext repeating patterns, the ciphertext evidently leaks the information that the cipher is supposed to protect. As more high is the resolution of the image as more evident becomes the issue.
The very same issue can be found in any bit-string containing repeated patterns.

The following is the very same message but encrypted using AES in CBC mode.



Any type of pattern is lost because of the avalanche effect implemented by CBC mode.
ECB excluded, the very same effect is given by any other mode of operation we saw so far.

More about this experiment implementation at the following link:

- https://blog.filippo.io/the-ecb-penguin/

Stream Ciphers

A stream cipher is a **symmetric** cipher where the plaintext digits are combined with a **pseudorandom** bit stream known as **keystream**.

In particular, each plaintext *i*-th bit is encrypted by XORing it with the *i*-th bit of the keystream.

The keystream is typically generated from a Cryptographically Secure Pseudo Random Number Generator **(CSPRNG)** such as a **Non-Lnear Feedback Shift Register** (**NLFSR**), a shift register whose input bit is a non-linear function of its previous state.

Most modern stream ciphers keystream generators are designed by combining one or more NLFSR generally of different lengths and with different feedback combinations (polynomials).

In a **synchronous** stream cipher the keystream is generated independently of the plaintext and ciphertext message. If a digit is added or removed from the message, synchronization is lost.

In a **self-synchronizing** stream ciphers *N* of the previous ciphertext digits are used to compute the keystream. The result is that the receiver automatically synchronize with the keystream generator after receiving *N* ciphertext digits, making easier to recover if digits are dropped or added to the stream.

Trivium

A simple synchronous stream cipher designed to provide a reasonable trade-off between hardware gate count and software implementation speed.

Trivium was submitted to the Profile II (hardware) of **eSTREAM** competition by **Christophe De Canniere** and **Bart Preneel** in **2004**.

Technical Overview

It generates up to 2^{64} bits of key stream from an 80-bit secret key and an 80-bit Initialization Vector.

It is based on the combination of three NLFSR of length 93, 84 and 110 bits.

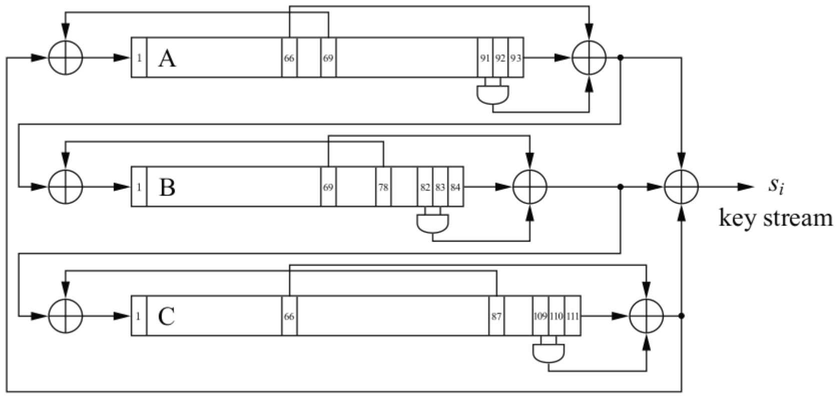
The output of each register is connected to the input of another register. Thus, the registers are arranged in circle-like fashion. The cipher can be viewed as consisting of one circular register with a total length of 93+84+111=288 bits.

The **input** of each register is computed as the XOR-sum of two bits:

- The output bit of the previous register.
- One register bit at a specific location is fed back to the input (e.g. bit 69 of A is fed back).

The **output** of each register is computed as the XOR-sum of three bits:

- The rightmost register bit (as with LFSR).
- One register bit at a specific location is fed forward to the output (e.g. bit 66 of A is fed forward).
- The output of a logical AND function whose input is two specific register bits.



The overall output is the XOR of the output of the three registers.

The feed forward paths involving the AND operation are crucial for the security of Trivium as they are the only non-linear component of the construction.

API

Even though the cipher definition works bit-wise, the implementation uses a more software friendly byte-wise approach.

Initialization

Zero out the context.

```
void cry_trivium_init(cry_trivium_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_trivium_clear(cry_trivium_ctx *ctx);
```

Key set
Set cipher key. Maximum length is 10 bytes, shorter keys are zero padded.

```
void cry_trivium_key_set(cry_trivium_ctx *ctx, const unsigned char *key, size_t size);
```

IV set
Initialization Vector set. Setting a new initialization vector resets the cipher, the key is maintained between resets.
Maximum iv length is 10 bytes; shorter IVs are zero padded.

```
void cry_trivium_iv_set(cry_trivium_ctx *ctx, const unsigned char *iv, size_t size);
```

Encrypt/Decrypt
The same function is used to encrypt and decrypt data (involution).

```
void cry_trivium_crypt(cry_trivium_ctx *ctx, unsigned char *dst,
                       const unsigned char *src, size_t size);
```

Two convenience encrypt/decrypt macro definitions

```
#define cry_trivium_encrypt cry_trivium_crypt
#define cry_trivium_decrypt cry_trivium_crypt
```

Usage example:

```
cry_trivium_ctx tri;                         /* Trivium context */
unsigned char *key = "123456789A";           /* 80-bit encryption key */
unsigned char *iv  = "abcdefghij";           /* 80-bit initialization vector */
unsigned char *msg = "Hello";                /* Message */
unsigned char buf[5];                        /* Encryption/decryption buffer */

cry_trivium_init(&tri);                      /* Context initialization */
cry_trivium_key_set(&tri, key, 10);          /* Key set */
cry_trivium_iv_set(&tri, iv, 10);            /* IV set */
cry_trivium_encrypt(&tri, buf, msg, 5);      /* Encrypt */
cry_trivium_decrypt(&tri, buf, buf, 5);      /* Decrypt (output stored in input buffer) */
cry_trivium_clear(&tri);                     /* Context cleanup */
```

ARC4

RC4 is a **byte oriented** stream cipher designed by **Ron Rivest** for RSA Security in **1987**.

RC4 is not an open standard and the details of how it works have never been officially published. In September 1994 a detailed description of the algorithm has been anonymously posted to the *Cypherpunks* mailing list. The leaked code was confirmed to be genuine as its output was found to match that of proprietary software using licensed RC4. Because the name RC4 is trademarked, it is often referred to as *ARCFOUR* or ***ARC4*** (meaning *alleged* RC4).

The cipher has gained immense popularity for its simplicity and performances, which has also made it widely accepted for numerous software applications.

Unfortunately in 2013 multiple vulnerabilities have been discovered in ARC4, rendering it insecure. It is especially vulnerable when the beginning of the output keystream is not discarded, or when nonrandom or related keys are used.

Technical Overview

The design of RC4 avoids the use of feedback shift registers and adopts a more software friendly design strategy which requires only bytes manipulations.

To generate the keystream, with a period greater than 10^{100}, the cipher makes use of an internal state consisting of two parts:

- A permutation of all 256 possible bytes values (denoted as “S”).
- Two 8-bit index pointers (denoted i and j).

A 256-byte key schedule is initially computed from the key, which can have a maximum length of 256 bytes. After that, each byte of the plaintext encrypted by XORing it with one byte of the key schedule after permuting the key schedule. This procedure is iterated until the plaintext is completely encrypted.

An initialization vector may be concatenated to the key.

The algorithm behind ARC4 is simple enough to allow the overall pseudocode to be reported here. The first part, the key schedule, is performed once at key-set time, and shall be done before the encryption/decryption phase.

Key schedule

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

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

After 256 such iterations, the S array is completely permuted, with each ordinal from 0 to 255 appearing once and only once.

Encryption/decryption :

```
i = j = 0;
for k = 0 to inputlen do
    i = (i + 1) mod 256;
    j = (j + S[i]) mod 256;
    swap(S[i], S[j]);
    output[k] = S[(S[i]+S[j]) mod 256] xor input[k];
```

API

Initialization

Zero out the context.

```
void cry_arc4_init(cry_arc4_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_arc4_init(cry_arc4_ctx *ctx);
```

Key set

Set cipher key. Maximum length is 255 bytes.

```
void cry_arc4_key_set(cry_arc4_ctx *ctx, const unsigned char *key, size_t size);
```

Encrypt/Decrpt

The same function is used to encrypt and decrypt data (involution).

```
void cry_arc4_crypt(cry_arc4_ctx *ctx, unsigned char *dst,
                    const unsigned char *src, size_t size);
```

Two convenience encrypt/decrypt macro definitions

```
#define cry_arc4_encrypt cry_arc4_crypt
#define cry_arc4_decrypt cry_arc4_crypt
```

Decrypt

Convenience wrapper, internally just calls the encrypt function.

```
void cry_arc4_decrypt(cry_arc4_ctx *ctx, unsigned char *dst,
                      const unsigned char *src, size_t size);
```

Usage example:

```
cry_arc4_ctx arc4;                               /* ARC4 context */
unsigned char *key = "123456789ABCDEFabcdef";    /* 168 bit key */
unsigned char *msg = "Hello";                    /* Message */
unsigned char buf[5];                            /* Encryption/decryption buffer */

cry_arc4_init(&arc4);                            /* Context initialize */
cry_arc4_key_set(&arc4, key, 21);                /* Key set */
cry_arc4_encrypt(&arc4, buf, msg, 5);            /* Encrypt */
cry_arc4_decrypt(&arc4, buf, buf, 5);            /* Decrypt (store output in input buffer) */
cry_arc4_clear(&arc4);                           /* Context cleanup */
```

Public Key Algorithms

A Public Key (PK) system is cryptographic system that uses a pair of different keys bounded by some mathematical property. For this reason is also known as **asymmetric** cryptosystem**.**

The general usage of a PK cryptosystem is to publish one of the keys, the **public key**, and keep one private, the **private key**. Everybody can encrypt a message using the public key, but only the private key owner (the recipient) will be able to decrypt the message.

Asymmetric cryptography is not meant to replace symmetric cryptography, mainly because it is several times slower than symmetric ciphers such as AES.

Today it finds its core applications in **digital signatures** and in symmetric cipher **key exchange**.

Confidentiality service

Data encrypted with the public key can be decrypted only with the corresponding private key.

$$D_{pvt}(E_{pub}(M))=M$$

Everyone can encrypt data with a public key, but only the key owner will be able to decrypt the information and recover the original message using the private key.

Its core application is to implement symmetric cipher **key exchange**.

Non-Repudiation service

Data encrypted with a private key can be decrypted only with the corresponding public key.

$$D_{pub}(E_{pvt}(M))=M$$

Only the private key owner is able to encrypt a message, but everyone is able to decrypt it.

Its core application is to implement **digital signature**.

Not every public key algorithm is able to provide both the services. There are algorithms explicitly constructed for digital signature and others only for ciphering.

Public-key algorithm **families** of practical relevance:

- **Integer factorization** schemes: based on large **integers** factorization problem. The most important representative is RSA.
- **Discrete logarithm** schemes: based on the discrete logarithm problem over **cyclic groups**. Important representatives are Diffie-Hellman, El Gamal, DSA,.
- **Elliptic Curve** schemes: generalization of discrete logarithm problem over elliptic curve groups. Important representatives are ECDSA, EC Diffie-Hellman.

Proofs of correctness of the schemes are all based on **number theory** well known theorems.

RSA

RSA is one of the first public-key cryptosystems and currently the most widespread one, designed in **1977** by **Ronald Rivest**, **Adi Shamir** and **Leonard Adleman**.

It is a **block cipher** in which the plaintext and ciphertext are treated as integers between 0 and *n*-1 for some block size *n*.

The underlying one-way function of RSA is the **integer factorization problem**: multiplying two large primes is computationally easy, but factoring the resulting product is very hard.

RSA is backed by a simple but well known **mathematical background** where the **Euler’s theorem** and **Euler’s phi** function play the most important roles.

Technical Details

The public key is represented by a couple of integers (e,n) and the private key by an integer (d).

Key generation procedure:

1. Choose two big integer primes *p* and *q*
2. Compute $n=p \cdot q$
3. Compute $\Phi(n)=(p-1)(q-1)$ (Euler totient)
4. Select the public exponent $e \in \{1,2,\ldots,\Phi(n)\}$ such that $gcd(e,\Phi(n))=1$
5. Compute the private key *d* such that $d \cdot e \equiv 1 \ mod \, \Phi(n)$

In practice *p* and *q* are very long numbers, usually 1024 bit long or more.

Encryption :

$$C=E_{pub}(M)=M^e mod\, n$$

Decryption:

$$M=D_{pvt}(M)=C^d mod\, n=(M^e)^d mod\, n$$

Proof of decryption correctness is not reported here but heavily replies on $gcd(e,\Phi(n))=1$.

Signature and **verification** procedures are equal to encryption and decryption, respectively, but the integer operations are performed by swapping the keys, i.e. to sign a message we encrypt it by using the private key *d* and to verify the signature we decrypt it using the public key *e* and we compare the result to the original message.

Schoolbook RSA

When used "*as-is*", RSA is called Schoolbook RSA and is proven to expose several weaknesses.

With respect to encryption:

- It is **malleable**: given the ciphertext $C=E_{pub}(M)=M^e \mod n$, anyone can compute $C'=C\cdot E_{pub}(k)=C\cdot k^e \mod n=(Mk)^e \mod n$. When the private key owner decrypts C' he will get Mk. In other words, **predictable changes** to ciphertexts can be performed.
- It is **deterministic**: if the message *M* is chosen from a small list of possible values, then it is possible to determine *M* from the ciphertext $C=E_{pub}(M)$ by simply encryping each possible value and comparing the result with *C*.
- Plaintext values $0, 1$ and $n-1$ produce ciphertexts equal to $0, 1$ and $n-1$.

With respect to digital signature:

- It is **malleable**: an attacker can combine signatures to create a new signature. For example, given a signature for the value 2 (i.e., $2^d \mod n$), it is possible to create a signature for 4 ($2^d\cdot 2^d \equiv 4^d \mod n$).
- Given the signatures of the message *kM* and *k*, for some constant k. Then we can get a valid signature for M by multiplying the first by the inverse of the second signature.
- Signature of $0, 1, n-1$ and $k^e \mod n$ is $0, 1, n-1$ and $k^e \mod n$.

Padding Schemes

To overcome to Schoolbook RSA issues, *RSA Security LLC* published four secure **usage schemes** in the **Public Key Cryptography Standard** (**PKCS**#1).

Before being encrypted, the cleartext is partitioned in blocks and each one is padded depending on the selected padding scheme.

There are two schemes for encryption:

- **PKCS v1.5** : random padding.
- **PKCS v2.1** (**OEAP**) : based on Optimal Asymmetric Encryption Padding scheme.

There are two schemes for digital signature:

- **PKCS v1.5** : fixed padding.
- **PKCS v2.1** (**PSS**): improved Probabilistic Signature Scheme.

API

Initialization

Zero out the context and set the padding scheme of choice.

```
int cry_rsa_init(cry_rsa_ctx *ctx, int padding);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_rsa_clear(cry_rsa_ctx *ctx);
```

Key set

Set the cipher key. The key size shall be equal to the key size of the backing cipher.

```
void cry_xxx_key_set(cry_xxx_ctx *ctx, const unsigned char *key, size_t size);
```

IV set

Set the Initialization Vector. Setting a new initialization vector resets the cipher, the key is maintained. The IV length is typically equal to the cipher block length (except GCM), shorter IV are zero padded.

```
void cry_xxx_iv_set(cry_xxx_ctx *ctx, const unsigned char *iv, size_t size);
```

Encrypt

Because of padding, ciphertext length is greater than cleartext length. Output buffer is malloced internally and its ownership is relinquished to the user (remember to free).

```
int cry_rsa_encrypt(cry_rsa_ctx *ctx, unsigned char **out, size_t *outlen,
                    const unsigned char *in, size_t inlen);
```

Decrypt

The same considerations apply as for the encrypt procedure.

```
int cry_rsa_decrypt(cry_rsa_ctx *ctx, unsigned char **out, size_t *outlen,
                    const unsigned char *in, size_t inlen);
```

Sign

The same considerations apply as for the encrypt procedure.

```
int cry_rsa_sign(cry_rsa_ctx *ctx, unsigned char **out, size_t *outlen,
                 const unsigned char *msg, size_t msglen);
```

Verify

Verification is performed internally by using the input message and the associated signature.

```
int cry_rsa_verify(cry_rsa_ctx *ctx, const unsigned char *sig, size_t siglen,
                   const unsigned char *msg, size_t msglen);
```

Key generation

Randomly generate RSA parameters and store them within context. Bits parameter refer to the bits of *n*.

```
int cry_rsa_keygen(cry_rsa_ctx *ctx, size_t bits);
```

Usage example:

```
cry_rsa_ctx rsa;
unsigned char *ciphertext, *cleartext, *sig;
size_t len;
const char *msg = "HelloWorld;

cry_rsa_init(&rsa, CRY_RSA_PADDING_PKCS_V15);   /* Initialize the context to use PKCS v1.5 /
cry_rsa_keygen(&rsa, 1024);                     /* Generate random RSA parameters */

cry_rsa_encrypt(&rsa, &ciphertext, &len, msg, strlen(msg)); /* Encrypt */
cry_rsa_decrypt(&rsa, &cleartext, &len, ciphertext, len);   /* Decrypt */

cry_rsa_sign(&rsa, &sig, &len, msg, msglen);                /* Sign */
cry_rsa_verify(&rsa, sig, len, msg, msglen);                /* Verify */

cry_rsa_clear(&rsa);  /* Cleanup */

/* Remember to release data malloced by the encrypt/decrypt/sign procedures */
free(ciphertext);
free(cleartext);
free(sig);
```

Diffie-Hellman

The Diffie Hellman Key Exchange (DHKE), proposed by **Whitfield Diffie** and **Martin Hellman** in **1976**, is a method used to securely exchange a secret over a public channel and was the first public-key protocol.

Technical Details

The security of Diffie-Hellman scheme relies on the computational intractability of finding solutions to the **discrete logarithm problem** (DLP) defined over **cyclic groups**.

The simplest implementation of the protocol uses the **multiplicative group of integers modulo *p***, where *p* is **prime** and *g* is a **primitive root** modulo *p*.

The basic idea behind DHKE is that exponentiation in $\mathbb{Z}_p$ is a one-way function (computationally infeasible to invert) and that exponentiation is commutative, i.e.

$$k \equiv (g^x)^y \equiv (g^y)^x \ mod\ p$$

Protocol:

1. Alice and Bob publicly agree to use the values of modulo *p* and base *g*.
2. Alice chooses a secret integer *a* and sends to Bob $A = g^a mod\ p$.
3. Bob chooses a secret integer *b* and sends to Alice $B = g^b mod\ p$.
4. Alice computes $s = B^a mod\ p = g^{ba} mod\ p$.
5. Bob computes $s = A^b mod\ p = g^{ab} mod\ p$.
6. Alice and Bob now share the secret *s*.

The **public key** is represented by the couple of integers (g, p) and the **private keys** are the integers (a) and (b) .

The protocol can be generalized to any finite **cyclic group**, such as **Elliptic Curves**.

API

Even though the DH key exchange procedure can be easily performed by directly using the MPI API, the library provides a DH structure to keep together the variables involved and some helper functions to help with the process.

Initialization
Zero out the context.

```
int cry_dh_init(cry_dh_ctx *ctx);
```

Cleanup
Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_dh_clear(cry_dh_ctx *ctx);
```

Set token
Save the remote peer token (e.g. if we are Alice, set Bob’s $B = g^b$).

```
int cry_dh_set_tok(cry_rsa_ctx *ctx, unsigned char *in, size_t len);
```

Get secret
Finalize and on get the shared secret (e.g. if we are Alice, get $s = B^a$).

```
int cry_dh_get_sec(cry_dh_ctx *dh, unsigned char *out, size_t len);
```

Usage example

Assuming we are Alice we first generate a random private key *a*, then we use the received *B* to compute the shared secret.

```
cry_dh_ctx dh;                                      /* DH Context */
unsigned char *p_str = "12df4d7689dff4c99d9ae57d7"; /* prime */
unsigned char *g_str = "1e32158a35e34d7b619657d6";  /* group generator */
unsigned char *e_str = "12187301a65d6dd67800f9368"; /* private key */
unsigned char B_raw[] = { 0xdf,0x62,0xf0,0xfd,0xa5,0x9c,0x8b,0x76,0xd3,0x25,0x04,0xab };
unsigned char sec_raw[13];

cry_dh_init(&dh);                          /* Context initialize */
cry_mpi_load_str(&dh.e, e_str);            /* Load private key */
cry_mpi_load_str(&dh.p, p_str);            /* Load prime */
cry_mpi_load_str(&dh.g, g_str);            /* Load generator */

cry_dh_set_tok(&dh, B_raw, sizeof(B_raw));   /* Set the received B value (as raw) */
cry_dh_get_sec(&dh, sec, sizeof(sec_raw));   /* Get the shared secret */
cry_dh_clear(&dh);                           /* Context cleanup */
```

Equivalent example by directly using the MPI library.

```
cry_mpi a, g, p, s;                         /* State variables */

cry_mpi_init_list(&a, &g, &p, &s, NULL);     /* Initialize state variables */
/* Assume that Alice correctly setted the parameters for: p, g, a (as above) */
cry_mpi_load_str(&s, B_str);                 /* Set the received B key /
cry_mpi_mod_exp(&s, &s, &a, &p);             /* Get the shared secret (modular exp) /
cry_mpi_store_bin(&s, sec_raw, sizeof(sec_raw)); /* Store as raw byte array */
cry_mpi_clear_list(&a, &g, &p, &s, NULL);    /* Clear state variables */
```

Elgamal Digital Signature

The Elgamal **signature scheme**, proposed by **Taher Elgama**l in **1985**, takes inspiration from the Diffie-Hellman key exchange algorithm and is also based on the difficulty of computing **discrete logarithms** over a **finite field**.

Unlike RSA, where encryption and digital signature are almost identical operations, the Elgamal digital signature is quite different from the encryption scheme with the same name.

Technical Details

Key generation

1. Choose a large prime p
2. Choose a primitive element α of $\mathbb{Z}_p^*$ or a subgroup of $\mathbb{Z}_p^*$
3. Choose a random integer $d \in \{2, 3, \ldots, p-2\}$
4. Compute $\beta = \alpha^d \, mod \, p$

The keys are now defined as: $k_{pub} = (p, \alpha, \beta)$ and $k_{pvt} = (d)$.

Given a message $M \in \{0, 1, \ldots, p-1\}$.

Signature:

1. Choose a random ephemeral key $k_E \in \{0, 1, 2, \ldots, p-2\}$ such that $gcd(k_E, p-1) = 1$.
2. Compute the signature parameters:

   $r \equiv \alpha^{k_E} mod \, p$

   $s \equiv (M - d \cdot r) k_E^{-1} mod \, (p-1)$

   The signature consists of the two integers *r* and *s*.

Verification:

1. Compute the value $t \equiv \beta^r \cdot r^s mod \, p$.
2. If $t \equiv \alpha^M mod \, p$ the signature is valid.

Verification Proof.

The signature is valid if

$$\alpha^M \equiv \beta^r \cdot r^s \equiv \alpha^{dr} \cdot \alpha^{k_E s} \equiv \alpha^{dr + k_E s} \; mod \, p$$

For Fermat’s little theorem the relationship holds if

$$M \equiv dr + k_E s \;\; mod \, (p-1)$$

From which the construction rule of the second signature parameter *s*.

API

Initialization

Zero out the context.

```
int cry_elgamal_init(cry_elgamal_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_elgamal_clear(cry_elgamal_ctx *ctx);
```

Signature

Sign a message. The *r* and *s* values are returned in the `cry_elgamal_sign` structure.
The signature length is two times the *p* length.

```
int cry_elgamal_sign(cry_elgamal_ctx *ctx, cry_elgamal_sig *sign,
                     const unsigned char *in, size_t len);
```

Verification

Verify a message signature. The *r* and *s* values are passed in the `cry_elgamal_sign` structure.

```
int cry_elgamal_verify(cry_elgamal_ctx *ctx, const cry_elgamal_sig *sign,
                       const unsigned char *in, size_t len);
```

Usage Example

```
cry_elgamal_ctx elg;                                /* Elgamal context */
unsigned char *p_str = "12df4d7689dff4c99d9ae57d7"; /* prime */
unsigned char *g_str = "1e32158a35e34d7b619657d6";  /* group generator */
unsigned char *d_str = "12187301a65d6dd67800f9368"; /* private key */
unsigned char *msg = "HelloWorld";                  /* Message */
unsigned char sign[26];                             /* Signature, double the length of p */

cry_elgamal_init(&elg);                             /* Initialize the context */
/* Load state parameters */
cry_mpi_load_str(&elg.p, 16, p_str);                /* Prime */
cry_mpi_load_str(&elg.g, 16, g_str);                /* Generator */
cry_mpi_load_str(&elg.d, 16, d_str);                /* Private key */
cry_mpi_mod_exp(&elg.y, &elg.g, &elg.d, &elg.p);    /* Public component: g^d mod p */

cry_elgamal_sign(&elg, sign, msg, strlen(msg));     /* Sign message */

/* Verify message */
res = cry_elgamal_verify(&elg, sign, input_raw, input_len); /* return 0 on success */

cry_elgamal_clear(&el);                              /* Cleanup */
```

DSA

Federal US government standard for **digital signatures** (DSS) proposed by **NIST** in **1991** and adoped as FIPS 186 in 1994.

Derived from the Elgamal signature scheme but with a fixed length 320-bit signature.

Technical Details

Key generation:

1. Generate a prime *p* with $2^{1023} < p < 2^{1024}$
2. Find a prime divisor *q* for *p*-1 with $2^{159} < q < 2^{160}$ ($p = qk + 1$)
3. Find an element α with $ord(\alpha) = q$ (α generates the subgroup with *q* elements)
4. Choose a random integer *d* with $0 < d < q$
5. Compute $\beta \equiv \alpha^d \, mod \, p$

The keys are now: $k_{pub} = (p, q, \alpha, \beta)$ and $k_{pvt} = (d)$

The central idea of DSA is that there are two cyclic groups involved. One is the large cyclic group Z_p^*, the order of which has bit length of 1024 bit, the second one is in the 160-bit subgroup of Z_p^*. This set-up yields shorter signatures.

In addition to the 1024-bit prime *p* and a 160-bit prime *q*, there are two other bit length combinations possible for the primes *p* and *q*. According to the latest version of the standard, the following combinations are allowed.

p	*q*	Signature
1024	160	320
2048	224	448
3072	256	512

If one of the other bit lengths is required, only Steps 1 and 2 of the key generation phase have to be adjusted accordingly.

Given a message $M \in \{0, 1, \ldots, p-1\}$

Signature:

1. Choose a random ephemeral key k_E with $0 < k_E < q$.
2. Compute $r \equiv (\alpha^{k_E} \, mod \, p) \, mod \, q$.

3. Compute $s \equiv (M + d \cdot r) \mod q$.

   The signature consists of the two integers *r* and *s*.

According to the standard the message *M* shall be the output of the SHA-1 hash after been applied to the original message.

Verification:

1. Compute $w \equiv s^{-1} \mod q$.
2. Compute $u_1 \equiv w \cdot M \mod q$.
3. Compute $u_2 \equiv w \cdot r \mod q$.
4. Compute $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \mod p) \mod q$.
5. If $v \equiv r \mod q$ the signature is valid.

The verification proof is a bit involved, but very similar to the Elgamal one.

API

Initialization

Zero out the context.

```
int cry_dsa_init(cry_dsa_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_dsa_clear(cry_dsa_ctx *ctx);
```

Signature

Sign a message. The output *r* and *s* values are returned in the `cry_elgamal_sign` structure.

```
int cry_dsa_sign(cry_dsa_ctx *ctx, cry_dsa_sig *sign,
                 const unsigned char *in, size_t len);
```

Verification

Verify a message signature. The *r* and *s* values are passed in the `cry_elgamal_sign` structure.

```
int cry_dsa_verify(cry_dsa_ctx *ctx, const cry_elgamal_sig *sign,
                   const unsigned char *in, size_t len);
```

Usage Example

```
cry_dsa_ctx dsa;

cry_dsa_init(&el);    /* Context initialization */
/* Load the algorith parameters: p, q, g, d, pub */
// ...
cry_dsa_sign(&dsa, &sign, sha1_raw, 20);               /* Sign  SHA1 of a message */
res = cry_dsa_verify(&dsa, &sign, sha1_raw, sha1_len); /* Returns 0 on success */
cry_dsa_clear(&dsa);
```

Hash Functions

Hash functions compute a digest of an arbitrary long message as a short **fixed-length** bit-string.

Unlike all the other cryptographic primitives so far in this manual, hash functions **do not have a key**.

Hash functions are an essential part of digital signature schemes, message authentication codes and other applications such as key derivation and password obfuscation.

Merkle–Damgård Construction

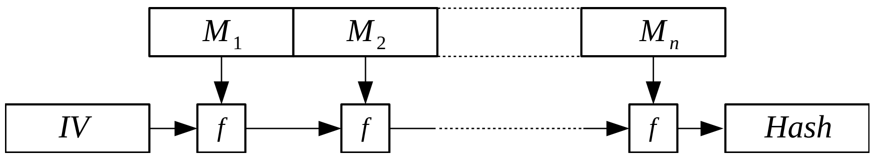
Most of today hash functions follows a design known as **Merkle–Damgård** construction.

With this design strategy, the message is sliced as an ordered set of *n* fixed-length chunks, with the last block padded up to a multiple of the chunk length.

Follows an iterated procedure where the current message chunk is feed to a **compression function** along with the previous step output to produce a new fixed length output.

The iteration stops when the last chunk has been processed, the last iteration output is the hash.

For the first iteration the input is the first message chunk and a fixed **Initialization Vector** dependent on the specific algorithm.



MD5

Message-Digest algorithm 5 is a cryptographic hash function designed by **Ronald Rivest** in **1992** to replace the earlier MD4 algorithm.

MD5 follows the Merkle-Damgård construction to produce a **128-bit digest** of a message with maximum length of 2^{64} bits.

In 2004 security researchers revealed a number of weaknesses in MD5 algorithm. The worst of them allows attackers to create multiple, differing input sources that, when the MD5 algorithm is used, result in the same output fingerprint (violation of **collision resistance** property).

Technical Details

The compression function processes the message in 512-bit blocks and consists of **four stages** of **16 rounds** each.

Padding

The message is padded so that its length is 64 bit short of being a multiple of 512. This padding is a single 1-bit added to the end of the message, followed by as many zeros are required. Then, a 64-bit representation of the original message length is appended to the result.

If the message length was already a multiple of block size, then a full padding block is appended.

Hash state

The hash state is held into a 128-bit buffer managed as four 32-bit words *A*, *B*, *C*, *D*.

The state initial values are:

A = 0x01234567, *B* = 0x89abcdef, *C* = 0xfedcba98, *D* = 0x76543210.

Hash Computation

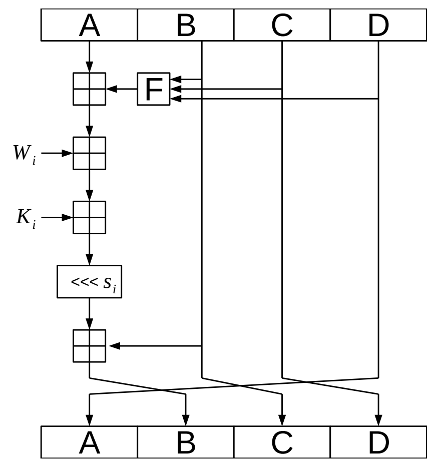
Each message block M_i is processed by 64 operations divided in four stages of 16 rounds each.

The algorithm uses a **message schedule** which computes a 32-bit word sequence $W_0, W_1, \dots, W_{63}$ for each of the 64 operations. Given m_k the *k*-th 32-bit word in the block M_i, the words W_j are derived as follows:

$$W_j = \begin{cases} m_j & 0 \le j \le 15 \quad (stage\,1) \\ m_{(5j+1) \bmod 16} & 16 \le j \le 31 \quad (stage\,2) \\ m_{(3j+1) \bmod 16} & 32 \le j \le 47 \quad (stage\,3) \\ m_{(7j+1) \bmod 16} & 48 \le j \le 63 \quad (stage\,4) \end{cases}$$

Each W_j has associated a 32-bit constant value K_j used as an addend and a rotation value s_i both used by the *j*-th operation step.

Picture of one arbitrary MD5 operation:



Where [+] is the 32-bit word addition, <<<*s* it the left rotation by *s* bits and *F* is a non-linear function. Note: each stage uses a different *F*.

API

Initialization

Zero out the context.

```
void cry_md5_init(cry_md5_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_md5_clear(cry_md5_ctx *ctx);
```

Update

Adds data to the hash state.

```
int cry_md5_update(cry_md5_ctx *ctx, const unsigned char *data, size_t len);
```

Finalize

Store the computed hash in the out buffer.

```
int cry_md5_digest(cry_md5_ctx *ctx, const unsigned char *out, size_t len);
```

Usage Example

```
cry_md5_ctx md5;                            /* MD5 context */
unsigned char *msg = "HelloWorld";          /* Message to be hashed */
unsigned char digest[16];                   /* Message hash buffer */

cry_md5_init(&md5);                         /* Context initialization */
cry_md5_update(&md5, msg, 3);               /* Add data to the hash state */
cry_md5_update(&md5, msg+3, strlen(msg)-3); /* Add more data to the hash state */
cry_md5_digest(&md5, digest, 16);           /* Finalization and get digest */
cry_md5_clear(&md5);                        /* Cleanup */
```

SHA-1

Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function designed by the **NSA** in 1995 based on the same principles of the MD5 algorithm.

SHA-1 follows a Merkle-Damgård construction and produces a **160-bit digest** of a message with a maximum length of 2^{64} bits.

Since 2005 SHA-1 has not been considered secure against well-founded opponents, and since 2010 many organizations recommended its replacement by SHA-2 or SHA-3.

Technical Details

The compression function processes the message in 512-bit chunks and consists of **four stages** of **20 rounds** each.

Padding

The message is padded so that its length is 64 bit short of being a multiple of 512. This padding is a single 1-bit added to the end of the message, followed by as many zeros are required. Then, a 64-bit representation of the original message length is appended to the result.

If the message length was already a multiple of block size, then a full padding block is appended.

Hash state

The hash state is held into a 160-bit buffer managed as five 32-bit words *A*, *B*, *C*, *D*, *E*.

The state initial values are:

A = 0x67452301, B = 0xEFCDAB89, C = 0x98BADCFE, D = 0x10325476, E = 0xC3D2E1F0.

Hash Computation

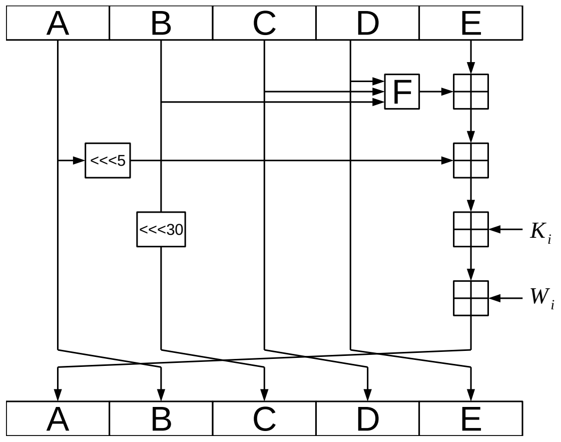
Each message block M_i is processed by 80 operation divided in four stages of 20 rounds each.

The algorithm uses a **message schedule** which computes a 32-bit word sequence $W_0, W_1, \ldots, W_{79}$ for each of the 80 operations. Given m_k the k-th 32-bit word in the block M_i, the words W_j are derived as follows:

$$W_j = \begin{cases} m_j & 0 \le j \le 15 \quad (stage\ 1) \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}) <<< 1 & 16 \le j \le 79 \quad (stages\ 2,3,4) \end{cases}$$

With $X <<< 1$ the left rotation of *X* by 1 bit position.

Picture of one arbitrary SHA-1 operation:



Where [+] is the 32-bit word addition, <<<*s* it the left rotation by *s* bits and *F* is a non-linear function. Note: each stage uses a different *F*.

API

Initialization

Zero out the context.

```
void cry_sha1_init(cry_sha1_ctx *ctx);
```

Cleanup

Zero out the context in a safe way (avoids compiler optimizations).

```
void cry_sha1_clear(cry_sha1_ctx *ctx);
```

Update

Adds data to the hash state.

```
int cry_sha1_update(cry_sha1_ctx *ctx, const unsigned char *data, size_t len);
```

Finalize
Store the computed hash in the out buffer.

```
int cry_sha1_digest(cry_sha1_ctx *ctx, const unsigned char *out, size_t len);
```

Usage Example

```
cry_sha1_ctx sha1;                               /* SHA-1 context */
unsigned char *msg = "HelloWorld";               /* Message to be hashed */
unsigned char digest[20];                        /* Message hash buffer */

cry_sha1_init(&sha1);                            /* Context initialization */
cry_sha1_update(&sha1, msg, 3);                  /* Add data to the hash state */
cry_sha1_update(&sha1, msg+3, strlen(msg)-3);    /* Add more data to the hash state */
cry_sha1_digest(&sha1, digest, 16);              /* Finalization and get digest */
cry_sha1_clear(&sha1);                           /* Cleanup */
```

Usage Example – Digital Signature

Assume that we have to sign a very long message by using RSA.

Because the message cannot be longer than the public key *p* parameter modulus, which we assume to be 1024-bit (128 bytes), we may intuitivelly think to sign the overall message by dividing it in 128 bytes chunks and sign them independently.

This naive approach has several problems:

1. High computational cost: RSA is expensive and here we are performing it over each chunk.
2. Long signature: production of *n* independent signatures.
3. Security limitations: since the signatures are independent, an attacked can remove or reorder the message chunks (along with the corresponding signatures) without the recipient can notice it.

Because of all these problems the generally adopted approach is to sign the message digest computed by an hash function with an output of at most of 128 bytes: $Sign(Hash(Message))$.

In this way the digital signature is compact, less expensive and protects the message integrity.