Hash Block Ciphers

M. Andrecut

2020

Calgary, Alberta, T3G 5Y8, Canada
mircea.andrecut@gmail.com

Abstract

Block ciphers are the most important building tools used in the design of secure communication systems based on symmetric cryptography. Here we discuss a class of block ciphers based only on cryptographically secure hash functions, and we provide a practical Python implementation.

Keywords: symmetric cryptography, block cipher, hash function

PACS: 89.20.Ff

1 Introduction

Ciphers are widely used cryptographic algorithms in the design of confidential communication systems for both data *encryption* and the corresponding *decryption*. In such a communication system, the confidentiality is achieved using the following protocol [1, 2]. The *sender* is encrypting the *plaintext* with an algorithm (cipher) and a secret *key*. The resulted encrypted (enciphered) information is called *ciphertext* and is sent to the *receiver*, which later can recover the plaintext also by using a secret key and the corresponding decryption algorithm. If the secret keys used for encryption and decryption are the same then the process is called *symmetric encryption*, and requires the sender and the receiver to share the secret key over a *secure channel* which cannot be eavesdropped (Fig. 1).

Let us denote by $\mathcal{X}$, $\mathcal{K}$, $\mathcal{Y}$ the sets of all messages, keys and respectively ciphertexts. Also, we assume that:

$$E(k,x) = y, \quad D(k,y) = x, \tag{1}$$

are the encryption and decryption functions, with $x \in \mathcal{X}$, $k \in \mathcal{K}$, and respectively $y \in \mathcal{Y}$. In order to define a cipher, the E and D functions must satisfy the following correctness property:

$$D(k, E(k,x)) = x, \ \forall x \in \mathcal{X}, \tag{2}$$

which means that D must be the inverse of E, $D = E^{-1}$.

Here we discuss a class of block ciphers based on cryptographically secure hash functions, and we show how the encryption and decryption functions can be implemented using only hash functions. Also, we provide a practical Python implementation of the proposed class.

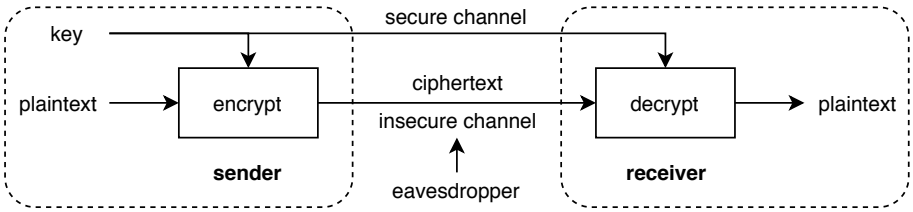

Figure 1: The symmetric encryption framework.

2 Perfect secrecy

A *perfect cipher* does not provide an attacker with any additional information about the plaintext, given the ciphertext [1, 2]. This means that for any two messages $x, x^* \in \mathcal{X}$, the probability that $x = x^*$ is:

$$\Pr[x = x^* | y = E(k, x)] = |\mathcal{X}|^{-1}, \tag{3}$$

where $|\mathcal{X}|$ is the size (cardinal) of $\mathcal{X}$.

Let us assume that $\mathcal{K} = \mathcal{X} = \mathcal{Y} = \{0,1\}^\ell$ are all the bit strings of length $\ell > 0$. The cipher with the encryption function:

$$E(k, x) = k \oplus x = y, \quad D(k, y) = k \oplus y = x, \tag{4}$$

is called *one-time pad* [1–3]. Here, $\oplus$ is the bit-wise exclusive-or operator:

$$\oplus : \{0,1\} \to \{0,1\}, \quad a \oplus b = (a + b) \mathrm{mod} 2, \tag{5}$$

such that $\langle \{0,1\}^n, \oplus \rangle$ is an Abelian group.

One can easily prove that the one-time pad is a perfect cipher [1–3]. Thus, if the keys are truly random (and therefore unpredictable by an attacker), and they are used only once, the attacker learns nothing about the plaintext when they are given a ciphertext. However, while the one-time pad is a perfect cipher, it is also impractical because the keys have to be as long as the messages, requiring a secure exchange prior communication, and then the keys can be used only once.

3 Block ciphers

A block cipher encrypts blocks of plaintext of a fixed length, typically 128 or 256 bits [1, 2]. That is, the plaintext is represented as a concatenation ($\|$) of blocks:

$$x = x_0 \,\|\, x_1 \,\|\, ... \,\|\, x_{L-1}. \tag{6}$$

Each block x_i has the same length ℓ, called the *block size*:

$$x_i = x_{i,0} x_{i,1} ... x_{i,\ell-1}. \tag{7}$$

Also, the length q of the key k is called the *key size*. The encryption-decryption functions are:

$$E : \{0,1\}^q \times \{0,1\}^\ell \to \{0,1\}^\ell, E(k,x) = y, \tag{8}$$

$$D : \{0,1\}^q \times \{0,1\}^\ell \to \{0,1\}^\ell, D(k,y) = x. \tag{9}$$

Therefore, a block cipher is a *permutation* which maps every possible block to some other block, and the key determines exactly the mapping.

Most block ciphers are *iterated block ciphers* [1, 2], consisting in the iterative application of the same round function $F(k,x)$ for a certain number of times R (Fig. 2):

$$x_i^r = F(k^r, x_i^{r-1}), 1 \le r \le R, \tag{10}$$

where $x_i^0 \equiv x_i$ is the input plaintext block, x_i^r is the ciphertext in round r, and the $R-$round keys $(k^1, ..., k^R)$ are derived from the secret key k using a key scheduling algorithm. The function $F(k^r, x_i)$ is therefore a permutation of the set of $\ell-$bit strings $\{0,1\}^\ell$.

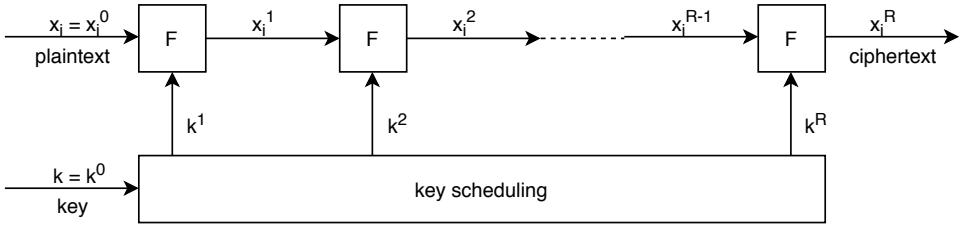

Figure 2: Typical structure of an iterated block cipher.

The round function F is constructed using the principles of *confusion* and *diffusion* [3]. The confusion step consists in establishing a very complex relation between the input-output-key bits, while the diffusion is used to hide the statistical properties of the plaintext. Two methods are frequently used in practice: the Feistel cipher and the Substitution Permutation (SP) network [1, 2].

The Feistel cipher was used in the 1970s for the Data Encryption Standard (DES) algorithm developed by IBM [4]. Since then, this architecture was analyzed extensively and using today's computers it can be easily broken with an exhaustive key-search attack, because the key size is only 56 bit. Therefore the plain DES is not suited in real world implementations. An alternative to DES is triple DES, or 3DES, which consists of three DES encryption steps. The 3DES has the advantage of using three different keys, which significantly extend the key space to 168 bits in order to counter the brute force attacks.

The Advanced Encryption Standard (AES) is based on the SP networks architecture, and it is the most common block cipher in current use [5]. AES has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. AES is also secure, and currently there are no practical attacks known against AES.

One of the most common mode of operation for block ciphers is the Cipher Block Chaining (CBC) [6]. In the CBC mode the successive plaintext blocks are linked using the exclusive-or operator (Fig. 3). For this process an *initialization vector* (v) is used as a predecessor for the first block. The initialization vector is a (pseudo)randomly generated bit string with the same block size ℓ, and it doesn't have to be secret, in fact it is typically added to the ciphertext, such that it can be later used in the decryption process. The initialization vector also allows the re-use of the same key for encryption, since it (pseudo)randomly modifies the plaintext before the encryption process.

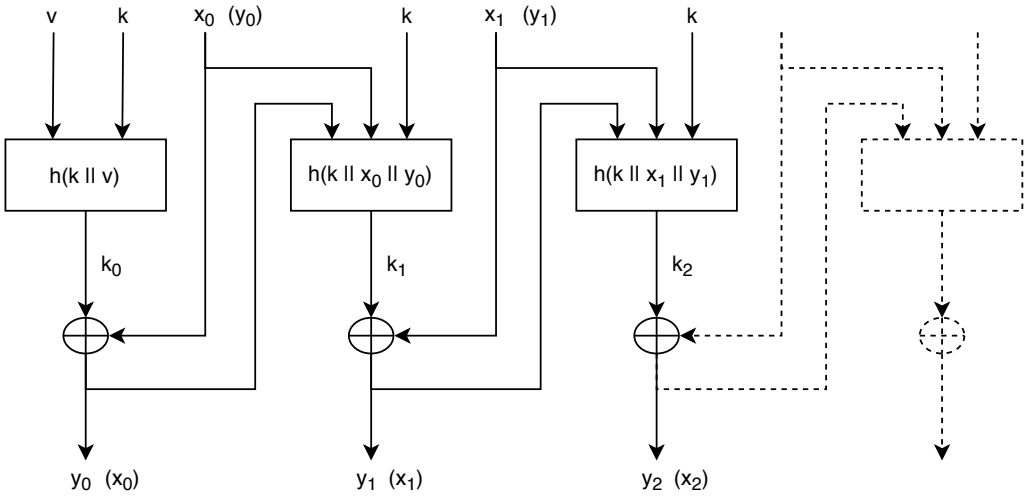

Figure 3: The HCBC mode of operation for hash block ciphers.

4 Hash block ciphers

The main idea behind the *hash block ciphers* is to combine the perfect one-time pad cipher with a *key expansion algorithm* based on *cryptographic hash functions*.

A cryptographic hash function $h : \{0,1\}^* \rightarrow \{0,1\}^\ell$ is designed to take a bitstring x of arbitrary finite length as input, and to output a bitstring $h(x)$ (called hash) of fixed length ℓ [1].

Typical hash functions that can be used for the hash block cipher approach are the standard SHA-256 and SHA-512 cryptographic hash functions, published by the NIST as a U.S. Federal Information Processing Standard (FIPS) [7]. For example using the SHA-512 function, no matter how big the input data is, the output will always have a fixed 512-bits length. Thus, using a cryptographic hash function one can easily verify that a given data maps to a given hash value, but if the input data is unknown it is impossible to reconstruct the data by only knowing the hash value. Here we will use this *one-way property* (irreversibility) of the cryptographic hash functions to devise a key expansion algorithm which then can be used together with the one-time pad to create a hash block cipher.

Assuming that the plaintext is organized as a concatenation of blocks of length ℓ, which is also equal to the cryptographic hash function output length, the encryption-decryption processes of the proposed *hash block cipher* are described by the following equations:

$$y_i = h(k \parallel x_{i-1} \parallel y_{i-1}) \oplus x_i, \quad y_0 = h(k \parallel v) \oplus x_0, \tag{11}$$

$$x_i = h(k \parallel x_{i-1} \parallel y_{i-1}) \oplus y_i, \quad x_0 = h(k \parallel v) \oplus y_0, \tag{12}$$

where $i = 1, ..., L-1$, $k \in \{0,1\}^\ell$ is the secret key and $v \in \{0,1\}^\ell$ is the initialization vector, having the same role as in the standard block ciphers.

Due to the intrinsic use of the one-time pad cipher, combined with the hash key expansion mechanism, the resulted Hash Cipher Block Chaining (HCBC) mode of operation is symmetric, and it just requires switching between plaintext and ciphertext in order to encrypt or decrypt the input data (Fig. 4). Thus, the computation in the hash block cipher can be implemented in-place, with y_i replacing x_i in the decryption process.

One can see that by chaining the hash computation we obtain the same key expansion in both encryption and decryption processes:

$$k_i = h(k \parallel x_{i-1} \parallel y_{i-1}), \quad k_0 = h(k \parallel v). \tag{13}$$

Due to the avalanche effect of the one-way cryptographic hash function this expansion mechanism will generate (pseudo)random keys which can be used in combinations with the one-time pad cipher. That is, even a single bit change in the argument of the hash function will (pseudo)randomly trigger half of the output bits to flip, generating a new (pseudo)random key which then can be used in the current encryption-decryption step. While this process is deterministic, the one-way property of the hash function does not allow the attacker to learn anything about the secret key k. The encryption key for the first block is $k_0 = h(k \parallel v)$, therefore if one always uses a (pseudo)randomly generated initialization vector v the attacker cannot learn anything about the secret key k.

One can also use a Hash Message Authentication Code (HMAC) to provide integrity and authenticity of the encrypted data. The HMAC is itself a cryptographic hash function h_{MAC}, which can be used to generate a MAC hash value of the ciphertext [1]. An important property of a MAC is that it is impossible to compute the MAC hash value without knowing the secret key. Therefore, if the ciphertext comes with a correct MAC attached, it means it has not been modified, and therefore the MAC provides ciphertext integrity. In the same time, a MAC also provides authenticity, since it is a signature generated with a secret key, such that it guarantees that the ciphertext was computed by the sender which is in the possession of the correct secret key. Therefore, the receiver can verify immediately the authenticity and integrity of the ciphertext.

The h_{MAC} requires a secret key which is recommended to be different from the secret key used in the encryption process. A possible solution is to (pseudo)randomly generate a long enough secret key k, and then to split the key in two $k = (k', k")$, such that the first half k' is used for the encryption process and the second half $k"$ is used for the HMAC process. Thus, if the key k has 64 bytes (512 bits), the encryption and the HMAC keys k' and $k"$ will both have a length of 256 bits, which is enough to guarantee good key security. The MAC hash value of the ciphertext is calculated as $h_{MAC} = h_{MAC}(k", y)$, and it is appended to the ciphertext submitted to the receiver, together with the initialization vector:

$$y_0 \parallel y_1 \parallel ... \parallel y_{L-1} \parallel v \parallel h_{MAC}. \tag{14}$$

5 Practical implementation

Here we discuss a practical Python implementation of the proposed *hash block ciphers* using the `sha512` and `blake2b` cryptographic hash functions provided by the standard `hashlib` module. The `blake2b` hash function provides an output of 512 bits and a security superior to SHA-2 and similar to that of SHA-3: immunity to length extension, indifferentiability from a random oracle (see Ref. [9] for details).

The implementation consists of a Python module `hcrypto.py` containing the class `Hcrypto` with the following methods:

- Key generation: `generate_key()`. Returns a 64 byte long secret random key, encoded in url safe base64.
- Data encryption (`data` must be in byte format): `sha512_encrypt(data)`, `blake2b_encrypt(data)`.
- Data decryption (`data` must be in byte format): `sha512_decrypt(data)`, `blake2b_decrypt(data)`.

Here is a simple example using the `BLAKE2b` function:

```
from hcrypto import Hcrypto

plaintext = '''Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua.'''

key = Hcrypto.generate_key()

H = Hcrypto(key)

ciphertext = H.blake2b_encrypt(plaintext.encode())

plaintext = H.blake2b_decrypt(ciphertext)

print('key=\n', key.decode(), '\n')

print('ciphertext=\n', ciphertext.decode(), '\n')

print('plaintext=\n', plaintext.decode(), '\n')
```

```
key=
qMWz1wCkkhcNyW0FgQQXrIKXovs7QYX4FU5UQJin6cj1Zjqk-6VKh4tUyCABsH7-jELqIYyzff4Img3Zb5Y4PQ==

ciphertext=
AH9iQ5qWFtHa1waQGMIe1OMhrNSpnQPUv3hztwRcvYzesDtmM6zqoAkQW5RshjjjK7-Fy3cmw-JpXx-CqzgiViSl
rLSIzpkVabKc-4g2vyD6Pwu3X5bL9tGZIPZI2q8jb-RVdtpX2dwskm4Epz8QgRIOfcm-uzIWt3JZA7Zgn9e78bj2
z3iHAUUkusZrESIJ8iVyoydzuHO72J50jUWhsDagaCcgUx2j6RjzznnjKU-TIAGHqa3ogatws0uihHQwpYsXJPjx
JTHere0ynvtgDLidThsx28Gvd6QhYxwzzbwdXS7ONMhBOF2w6AO95fVKRGRmeJWp9f8GJlvuLTg=

plaintext=
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua.
```

Here we provide the Python module `hcrypto.py` with the class `Hcrypto`.

```
import os
import hmac
import base64
from hashlib import sha512
from hashlib import blake2b

class Hcrypto(object):

    def __init__(self, key):
```

```
    key = base64.urlsafe_b64decode(key)
    if len(key) != 64:
        raise ValueError("key must be 64 url-safe base64-encoded bytes!")
    self._block = 64
    self._ekey = key[:32]
    self._skey = key[32:]

@classmethod
def generate_key(cls):
    return base64.urlsafe_b64encode(os.urandom(64))

def _xor(self, x, y):
    return bytes([a^b for (a,b) in zip(x, y)])

def sha512_encrypt(self, data):
    if len(data) == 0:
        raise ValueError("data is empty!")
    data = bytearray(data)
    v = os.urandom(self._block)
    h = sha512(self._ekey + v).digest()
    N, R, n = int(len(data)/self._block), len(data) % self._block, 0
    while n < N:
        nB, nBB = n*self._block, (n+1)*self._block
        d = bytes(data[nB:nBB])
        data[nB:nBB] = self._xor(d, h)
        h = sha512(self._ekey + data[nB:nBB] + d).digest()
        n += 1
    if R > 0:
        NBR = N*self._block+R
        data[NBR-R:NBR] = self._xor(data[NBR-R:NBR], h[0:R])
    data = data + v
    h = hmac.new(self._skey, data, sha512)
    data = data + h.digest()
    return base64.urlsafe_b64encode(data)

def sha512_decrypt(self, data):
    data = base64.urlsafe_b64decode(data)
    h = hmac.new(self._skey, data[:-self._block], sha512)
    if not hmac.compare_digest(data[-self._block:], h.digest()):
        raise ValueError("hmac failed!")
    v = data[-2*self._block:-self._block]
    data = bytearray(data[:-2*self._block])
    h = sha512(self._ekey + v).digest()
```

```
    N, R, n = int(len(data)/self._block), len(data) % self._block, 0
    while n < N:
        nB, nBB = n*self._block, (n+1)*self._block
        d = bytes(data[nB:nBB])
        data[nB:nBB] = self._xor(d, h)
        h = sha512(self._ekey + d + data[nB:nBB]).digest()
        n += 1
    if R > 0:
        NBR = N*self._block+R
        data[NBR-R:NBR] = self._xor(data[NBR-R:NBR], h[0:R])
    return bytes(data)

def blake2b_encrypt(self, data):
    if len(data) == 0:
        raise ValueError("data is empty!")
    data = bytearray(data)
    v = os.urandom(self._block)
    h = blake2b(self._ekey + v).digest()
    N, R, n = int(len(data)/self._block), len(data) % self._block, 0
    while n < N:
        nB, nBB = n*self._block, (n+1)*self._block
        d = bytes(data[nB:nBB])
        data[nB:nBB] = self._xor(d, h)
        h = blake2b(self._ekey + data[nB:nBB] + d).digest()
        n += 1
    if R > 0:
        NBR = N*self._block+R
        data[NBR-R:NBR] = self._xor(data[NBR-R:NBR], h[0:R])
    data = data + v
    h = hmac.new(self._skey, data, blake2b)
    data = data + h.digest()
    return base64.urlsafe_b64encode(data)

def blake2b_decrypt(self, data):
    data = base64.urlsafe_b64decode(data)
    h = hmac.new(self._skey, data[:-self._block], blake2b)
    if not hmac.compare_digest(data[-self._block:], h.digest()):
        raise ValueError("hmac failed!")
    v = data[-2*self._block:-self._block]
    data = bytearray(data[:-2*self._block])
    h = blake2b(self._ekey + v).digest()
    N, R, n = int(len(data)/self._block), len(data) % self._block, 0
    while n < N:
```

```
            nB, nBB = n*self._block, (n+1)*self._block
            d = bytes(data[nB:nBB])
            data[nB:nBB] = self._xor(d, h)
            h = blake2b(self._ekey + d + data[nB:nBB]).digest()
            n += 1
        if R > 0:
            NBR = N*self._block+R
            data[NBR-R:NBR] = self._xor(data[NBR-R:NBR], h[0:R])
        return bytes(data)
```

6 Conclusion

We have discussed a class of block ciphers based on cryptographically secure hash functions, as an alternative to the frequently used ciphers, like AES for example. Also, we have provided a practical Python implementation of the proposed class.

7 Disclaimer

Warning, "do it yourself encryption" is not encouraged, better use an existing validated approach. The method described here was developed only for educational purposes, it is not fully tested and validated, in practice use at your own risk.

References

[1] Stinson D. R., Cryptography Theory and Practice, 3rd ed., CRC Press, Boca Raton, 2006.

[2] Katz J., Lindell Y., Introduction to modern cryptography, CRC Press, Boca Raton, 2008.

[3] Shannon C., Communication Theory of Secrecy Systems, Bell System Technical Journal, 28(4), 656 (1949). https://ieeexplore.ieee.org/document/6769090

[4] Data Encryption Standard, Federal Information Processing Standard PUB, USA: https://csrc.nist.gov/CSRC/media/Publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf

[5] Advanced Encryption Standard, United States National Institute of Standards and Technology (NIST): https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[6] Ehrsam W.F., Meyer H.W., Smith J.L., Tuchman W.L., Message verification and transmission error detection by block chaining, US Patent 4074066, 1976. https://patents.google.com/patent/US4074066A/en

[7] United States National Institute of Standards and Technology (NIST): Descriptions of SHA-256, SHA-384, and SHA-512, http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf

[8] United States National Institute of Standards and Technology (NIST): Draft sha-3 standard: Permutation-based hash and extendable-output functions, FIPS 202, August 2015. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf.

[9] Aumasson J-.P., Meier W., Phan R.C.-W., Henzen L., The Hash Function BLAKE, Springer, 2015. https://www.springer.com/gp/book/9783662447567