AES Block Cipher

Blockciphers are central tool in the design of protocols for shared-key cryptography

What is a blockcipher?

It is a function E of parameters k and n that maps $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$

The function E takes two inputs, a k-bit string (key) and an n-bit string (plaintext), and returns an n-bit string (ciphertext)

For each key $K \in \{0, 1\}^k$, we let E_K: $(\{0, 1\}^n \rightarrow \{0, 1\}^n)$ be the function defined by $E_K(M) = E(K, M)$

For any blockcipher and any key K, it is required that the function E_K be a **permutation** on $\{0, 1\}^n$ (it is a *bijection* -- one-to-one and onto function)

Bijection indicates that for every $C \in \{0, 1\}^n$, there is exactly one $M \in \{0, 1\}^n$ such that $E_K(M) = C$

AES Block Cipher

E_K has an inverse, denoted E_K^{-1}, that also maps $\{0, 1\}^n$ to $\{0, 1\}^n$ with $E_K^{-1}(E_K(M)$ $= M$ and $E_K^{-1}(E_K(C) = C$ for all M, C in $\{0, 1\}^n$

We let E^{-1}: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be defined by $E^{-1}(K, C) = E_K^{-1}(C)$

This is the inverse blockcipher of E

Properties:

- The blockcipher E should be a publicly specified algorithm
- Both the cipher E and its inverse E^{-1} should be easily computable

In a typical use, a random key K is chosen and kept **secret** between a pair of users

The function E_K is used by both parties to process data to be exchanged

We assume that the adversary **will be able** to obtain some *input-output* examples of E_K, i.e., pairs of the form (M,C) where $C = E_K(M)$, but will not have the key K

Therefore, goal of the adversary is to recover key K with the input/output examples

AES Block Cipher

In 1998, NIST announced a competition for a new blockcipher to replace DES

AES addresses short 2^{56} key length, software speed, block size (64 ->128) of DES

Fifteen algorithms were submitted to NIST, second round narrowed number of five -- in summer of 2001, NIST announced that algorithm called Rijndael won

Authors are from Belgium, Joan Daemen and Vincent Rijmen

```
function AES_K(M)
    (K_0, ..., K_10) <- expand(K)
    s <- M XOR K_0
    for r = 1 to 10 do
        s' <- SBOX(s)
        s* <- shift-rows(s')
        if r <= 9 then s+ <- mix-cols(s*) else s+ <- s* fi
        s <- s+ XOR K_r
    endfor
    return s
```

AES

AES has a block length of $n = 128$ bits, and a key length k that is variable, 128, 192 or 256 bits

AES can be explained in terms of four additional mappings: *expand*, *SBOX*, *shift-rows* and *mix-cols*

Expand takes a 128-bit string and produces a vector of 11 keys (K_0, ..., K_{10})

The other three functions **bijectively map** 128-bits to 128-bits

AES consists of 10 rounds, each identical except for the K_i used, and the omission of *mix-cols* in the 10th round

The operations of *SBOX* and *mix-cols* involve arithmetic on bytes

The arithmetic structure has all the properties necessary to be called a *finite field*

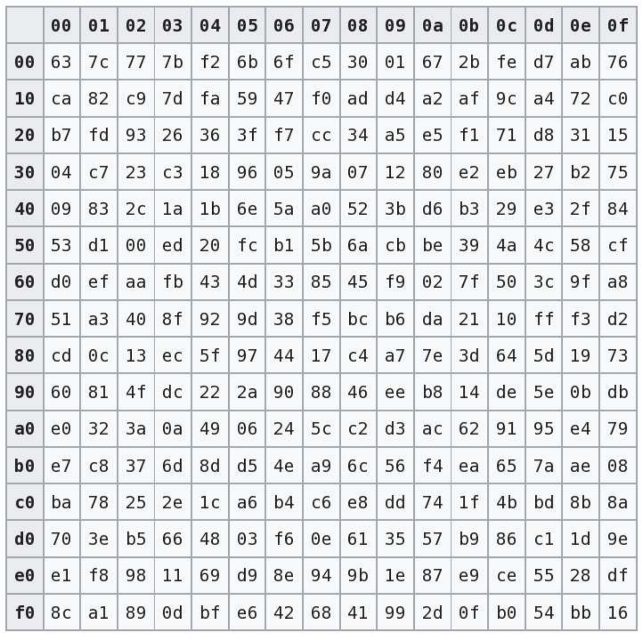
See http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

AES

SBOX: Each input byte $a_{i,j}$ is replaced with $SBOX(a_{i,j})$ using an 8-bit substitution box ($\{0, 1\}^8$ -> $\{0, 1\}^8$)

Low order 4 bits

High order 4 bits



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

This operation provides the **non-linearity** in the cipher

The SBOX is derived from the multiplicative inverse over GF(2^8), which is known to have good non-linearity properties

AES

Shift-rows: takes the 16 bytes of SBOX, $s_0 s_1 \ldots s_{15}$ and makes a 4 x 4 table

$$\begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix} \xrightarrow[\substack{\text{Rotate row 3 by 2 elements} \\ \text{Rotate row 4 by 3 elements}}]{\substack{\text{Rotate row 1 by 0 elements} \\ \text{Rotate row 2 by 1 elements}}} \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_5 & s_9 & s_{13} & s_1 \\ s_{10} & s_{14} & s_2 & s_6 \\ s_{15} & s_3 & s_7 & s_{11} \end{bmatrix}$$

This step prevents the columns from being **linearly independent**, in which case, AES degenerates into four independent block ciphers

Mix-cols:

Here the resulting columns in the 4 x 4 table above are combined using an invertible linear transformation

The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes

AES

Together with ShiftRows, MixColumns provides **diffusion** in the cipher

MixColumns multiplies each column by a fixed matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Matrix multiplication is composed of multiplication and addition (XOR) of the entries with the multiplication operation defined as follows:

Multiplication by 1 means no change

Multiplication by 2 means shifting 1-bit to the left

Multiplication by 3 means shifting 1-bit to the left and then performing XOR with the initial unshifted value

After shifting, a conditional XOR with 0x1B should be performed if the shifted value is larger than 0xFF -- these are special cases of multiplication in GF(2^8)

AES

In more general sense, each column is treated as a polynomial over GF(2^8):

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

And is then multiplied with a fixed polynomial:

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

And then taking the result modulo:

$$x^4 + 1$$

AddRoundKey: the subkey is combined with the output of Mix-cols

For each round, a subkey is derived from the main key using Rijndael's key schedule

```
s <- s+ XOR K_r
```

Next round ciphertext *s* is computed by bitwise XORing each byte of the Mix-cols output with the corresponding byte of the subkey

Note that in the first round, the plaintext *M* and original key K_0 is used to compute *s*

SHA-3 Secure Hash Algorithm

Block ciphers serve the *confidentiality* requirement of information security

As we discussed earlier, *authentication* and *data integrity* are important orthoganol properties of information security

Secure hash functions play a central role in serving these properties

Similar to encryption, the NIST standard for secure hash has changed over time

- In 2006, NIST organized the *NIST hash function competition* for SHA-3

- This was driven by concerns over the successful attacks on MD5 and SHA-0, and theoretical attacks on SHA-1

- SHA-3 is designed to supplement SHA-2 (not replace it)

51 candidates entered the competition in 2008, 14 were selected in July 2009 and a final set of 5 candidates were selected in Dec. 2010

Keccak Secure Hash Algorithm

Keccak won the competition on Oct. 2012

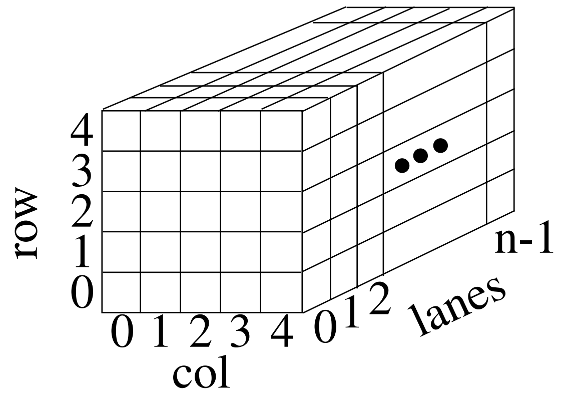
Keccak is a cryptographic hash function designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche

The SHA-3 standard was released by NIST on August 5, 2015

SHA-3 uses the *sponge construction*

In the first phase, data is **absorbed** into the sponge, which is later **squeezed** out

Absorbion involves XORing message blocks into the internal *state*, which is a large array of bits partitioned into 3 dimensions



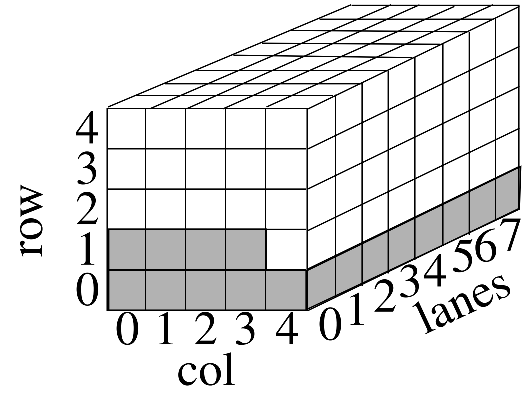
Row and column are always 5x5 in any version

The number of lanes can be configured as 1, 2, 4, 8, 16, 32, 64

Keccak-f[200] is the smallest version recommended, with lane size = 8

Keccak Secure Hash Algorithm

keccak-f[200] has 200 bits of internal state



Keccak has two parameters, *rate* and *capacity*

rate refers to the size of the message blocks while capacity is what remains

With *rate* equal to 72, 200-72 = 128 bit *capacity*

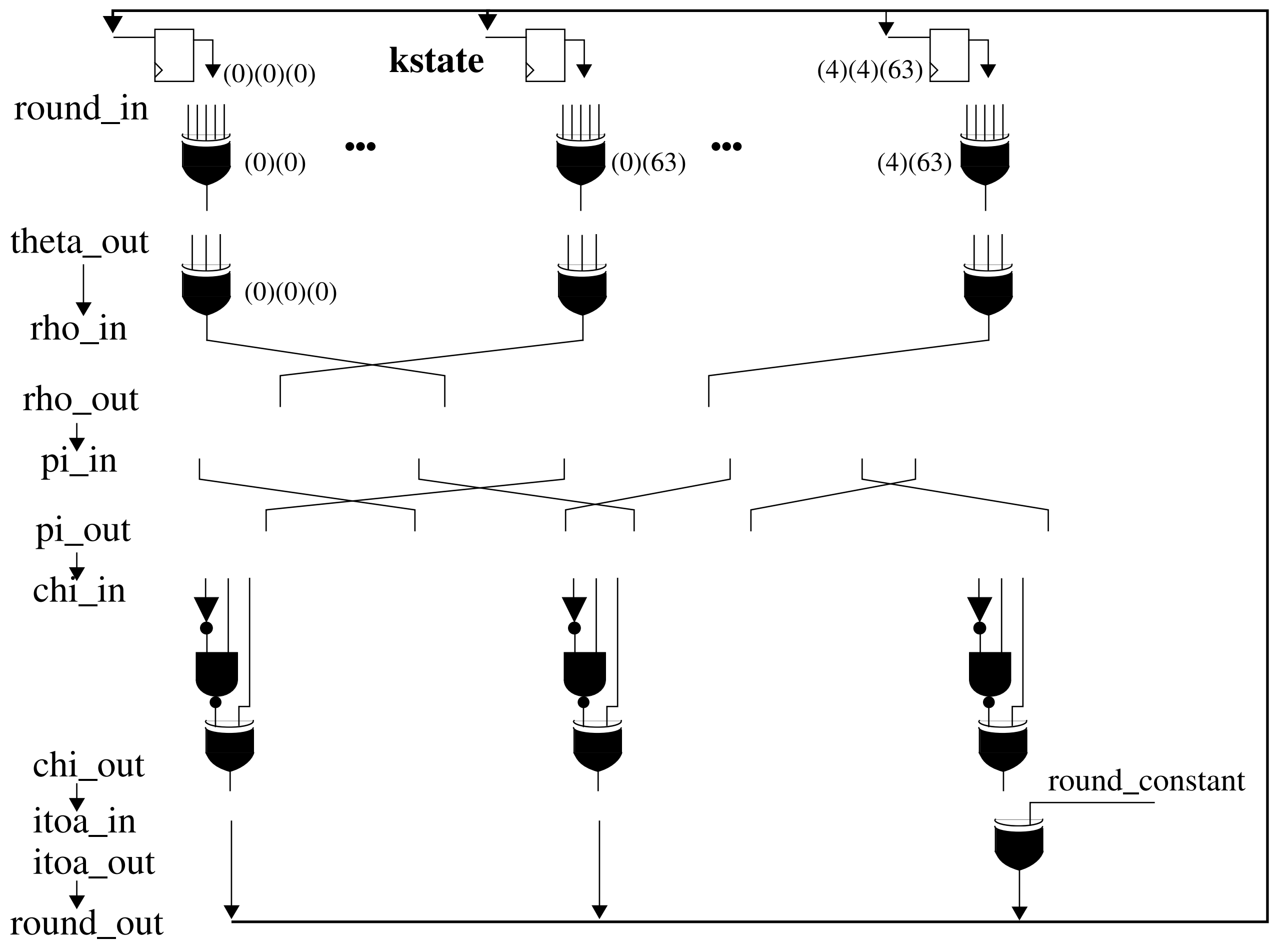
Capacity of 128 provides an equivalent security level of 64 bits

The message of 72-bits is XOR'ed into the 0-state in round 1

Keccak-f[200] hashes this string with the state in 18 rounds (*Keccak-f[1600]* has 24 rounds)

Keccak Secure Hash Algorithm

Each round manipulates the state (*kstate*) using the the following datapath operations



The algorithm is elegant and easily configured for different applications from high-security (*keccak-f[1600]*) to resource constrained (*keccak-f[200]*)

HMAC

HMAC is a keyed-hash message authentication code

(https://en.wikipedia.org/wiki/Hash-based_message_authentication_code)

HMAC leverages a cryptographic hash function, e.g. SHA-3, and can be used to verify data integrity and authenticity of a message

Commonly used terms include HMAC-MD5 and HMAC-SHA1, which use MD5 and SHA-1 cryptographic hash functions

Iterative hash functions, such as SHA-1, break the message into 512-bit blocks and **compress** the message into a smaller, e.g., 128-bit digest

Data integrity and authenticity is accomplished by transmitting the message (encrypted or un-encrypted) and the digest to the receiver

The receiver carries out the same process using her (shared) secret key on the received message to compute a second digest, which is compared with the received digest

If the digests match, then the message is authentic

HMAC

The most straightforward method for computing the ’tag’ (which is the hash digest) is

$$\text{MAC} = H(\text{key} \,\|\, message)$$

Unfortunately, this subjects most hash functions to *length extension attacks* (SHA-3 is naturally resistant to these attacks, making this construction secure -- see KMAC)

Here, the transmitted digest can be used by adversaries to create a new valid digest after adding malicious components to the original message

The digest is loaded into the hash function and run with the additional text

Notice that knowledge of the secret key is **not needed** in this attack because the original digest embodies the secret key

Instead, HMAC uses **two separate runs** of the cryptographic hash function and does **not** use the secret key directly

HMAC

HMAC derives an *inner* and *outer* key from the secret key and runs the hash twice:

- The first hash is carried out on the message and *inner* key
- The second hash uses the inner hash and *outer* key to produce the final **digest**

$$\text{HMAC}(K, m) = H((K'' \oplus \text{opad}) \,\|\, H((K' \oplus \text{ipad}) \,\|\, m))$$

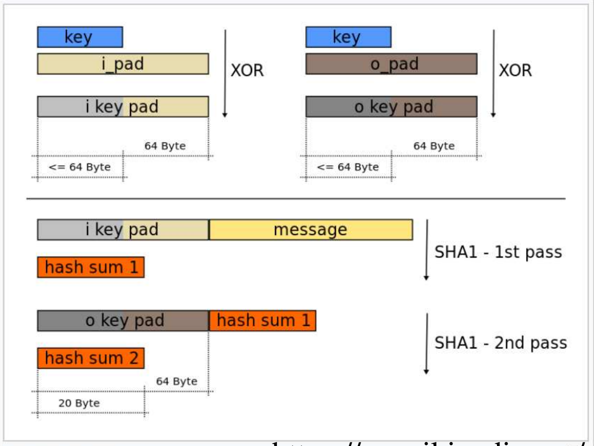
H is the hash function, *K* is the secret key and *m* is the message

K' is another secret key derived from *K* by either padding *K* to the right with extra zeros until it matches the hash input size, or hashing *K* if it is longer

|| represent concatenation and (+ with circle) is XOR

opad is the outer padding: 0x5c5c... (one block long)

ipad is the inner padding: 0x3636... (one block long)



HMAC-SHA1 generation. https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

Diffie-Hellman Key Exchange

Allows two parties with no prior knowledge of each other to jointly establish a **shared secret** key over an *insecure channel*

(https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#Description)

The shared key is then used to encrypt messages using a *symmetric* encryption scheme such as AES

Alice and Bob agree on a common starting color (this is not a secret) →

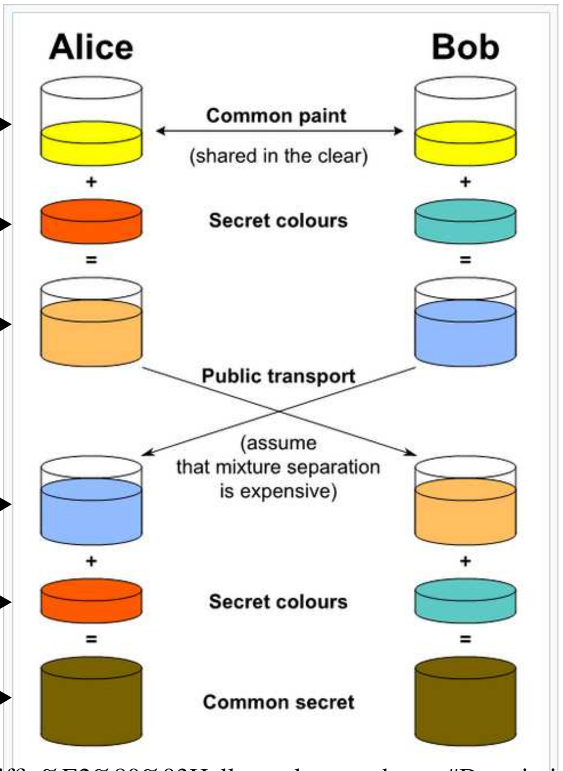
Each selects a color that they keep to themselves →

Secret colors are mixed with common color →

Mixed colors are exchanged →

Secret colors are mixed with exchanged colors →

Results in a shared secret color no one else knows →



https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#Description

Diffie-Hellman Key Exchange

Original scheme uses *multiplicative group of integers* **modulo** p

Where p is prime and g is a *primitive root* modulo p

Example (public values in blue and secret values in red)

- $p = 23$ and based $g = 5$ (which is a primitive root modulo 23)
- Alice choose a secret integer $a = 4$ and sends Bob $A = g^a \bmod p$

  $A = 5^4 \bmod 23 = 4$
- Bob chooses a secret integer $b = 3$ and sends Alice $B = g^b \bmod p$

  $B = 5^3 \bmod 23 = 10$
- Alice computes $s = B^a \bmod p$

  $s = 10^4 \bmod 23 = 18$
- Bob computes $s = A^b \bmod p$

  $s = 4^3 \bmod 23 = 18$

They now share the secret 18, which can be used as an symmetric encryption key

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$