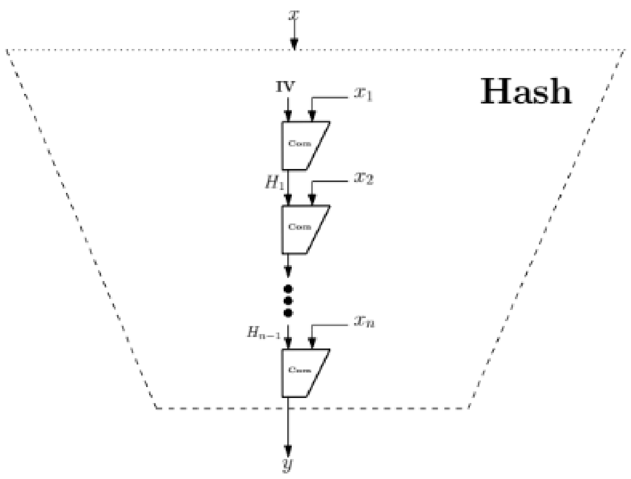
2021-04-22 theory

April 2021

7.2.1 *Compression & Merkle-Damgärd*

Merkle-Damgärd is a method to construct an **Hash** from a compression function **Com**. An *IV* initialization vector is necessary.
After a padding, the input x is divided in blocks $x = x_1x_2x \ldots x_n$ and it goes through the loop below:

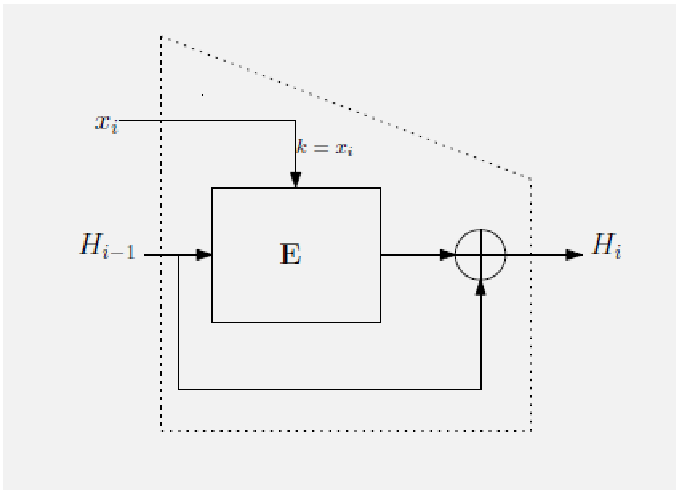


where each step i is composed of an operation made by the compression function, which receive as input:

- a digest H_{i-1} on n bits (at first iteration it's the IV)
- the input block x_i on b bits ($b > n$)

and it outputs an hash H_i on n bits.
The final result is y: a compressed digest of the input x.

The compression function can be implemented with a block cipher:
Davies-Mayer construnction of *Com* function
A block-cipher function *Enc* is used to get a compression function *Com*:



Observe that the blocks x_i are used as keys and the previous H_i's as blocks to be ciphered.
Notice the XOR at the end of the operation:

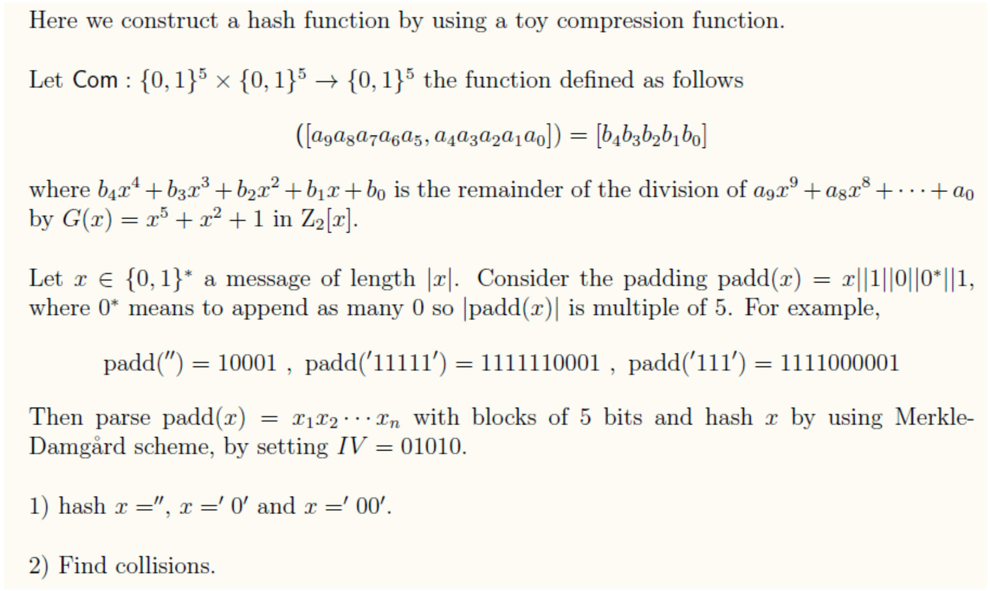
$$Enc_{x_i}(H_{i-1} \oplus H_{i-1})$$

By comparing it with Salsa20 and Chacha20, it is possible to notice that last XOR is used to implement a sort of key-whitening, as Salsa20 and Chacha20 do by inserting an additional ADD at the end of their iterations, in order to avoid recovering the key.

Exercise 7.2.4:*Show that David-Meyer function Com has fixed points. Namely, for any x_i there are H such that: $H = Enc(H) \oplus H$.*

Solution:
A notable property of the Davies–Meyer construction is that even if the underlying block cipher is totally secure, it is possible to compute fixed points for the construction: for any m, one can find a value of H such that $E_m(H) \oplus H = H$: one just has to set $H = E_m^{-1}(0)$. This is a property that random functions certainly do not have.

Exercise 7.2.5:



Here we construct a hash function by using a toy compression function.

Let Com : $\{0,1\}^5 \times \{0,1\}^5 \to \{0,1\}^5$ the function defined as follows

$$([a_9a_8a_7a_6a_5, a_4a_3a_2a_1a_0]) = [b_4b_3b_2b_1b_0]$$

where $b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$ is the remainder of the division of $a_9x^9 + a_8x^8 + \cdots + a_0$ by $G(x) = x^5 + x^2 + 1$ in $Z_2[x]$.

Let $x \in \{0,1\}^*$ a message of length $|x|$. Consider the padding $\mathrm{padd}(x) = x||1||0||0^*||1$, where 0^* means to append as many 0 so $|\mathrm{padd}(x)|$ is multiple of 5. For example,

$$\mathrm{padd}('') = 10001\ ,\ \mathrm{padd}('11111') = 1111110001\ ,\ \mathrm{padd}('111') = 1111000001$$

Then parse $\mathrm{padd}(x) = x_1x_2\cdots x_n$ with blocks of 5 bits and hash x by using Merkle-Damgård scheme, by setting $IV = 01010$.

1) hash $x ='',\ x =' 0'$ and $x =' 00'$.

2) Find collisions.

Solution:

1. a. ” $\to$ padd(”) $=$ 10001 $\to$ Just 1 round: $H(x) = Enc(Iv, 10001) \oplus IV = Com(01010, 10001) = 10110$

   b. ’0’ $\to$ padd(’0’) $=$ 10001 $\to$ As before.

   c. ’00’ $\to$ padd(’0’) $=$ 01001 $\to$ Just 1 round: $H = Enc(IV, 01001) \oplus IV = Com(01010, 00101) = 00010$

2. The compression function can be written as $f : \{0,1\}^{2\cdot 5} \to \{0,1\}^5$. The hash function is not preimage resistant (it means that for a given h in the output space of the hash function, it is not hard to find any message x with $H(x) = h$). Since $f(H_{i-1}, m_i) = Enc_{H_{i-1}}(m_i)$ one can easily obtain a preimage of a single block message using the decryption function. To obtain a second preimage or a collision, we use the same idea. Let $m = m_1||m_2$ be a string of length $2 \cdot 5$ and we want another string with the same hash value. Let $H_0 = IV$, $H_1 = Enc_{IV}(m_1)$, $H_2 = Enc_{H_1}(m_2)$. That is $h = m_1||m_2$ (note that we are already considering padding: "m_i" considers also padd() function).
   Now choose m' arbitrarily and set

$$m_2' = D_{E_{IV}(m_1')}(H_2).$$

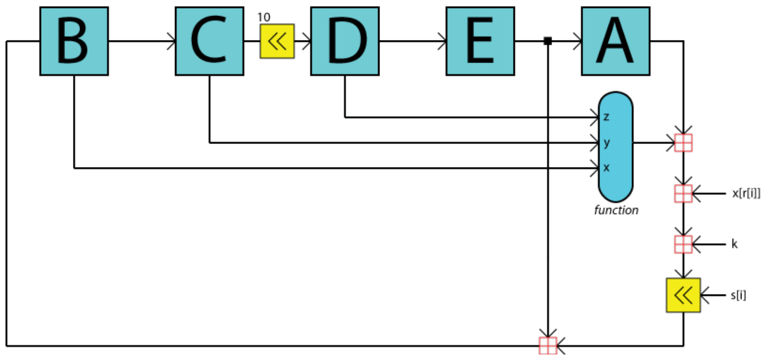
Then

$$\begin{aligned} h(m_1'||m_2') &= E_{H_1'}(m_2') = E_{E_{IV}(m_1')}(m_2') = \\ &= E_{E_{IV}(m_1')}(D_{E_{IV}(m_1')}(H_2)) = \\ &= H_2 = h(m_1||m_2). \end{aligned}$$

An example of *Merkle-Damgärd* application: RIPEMD-160

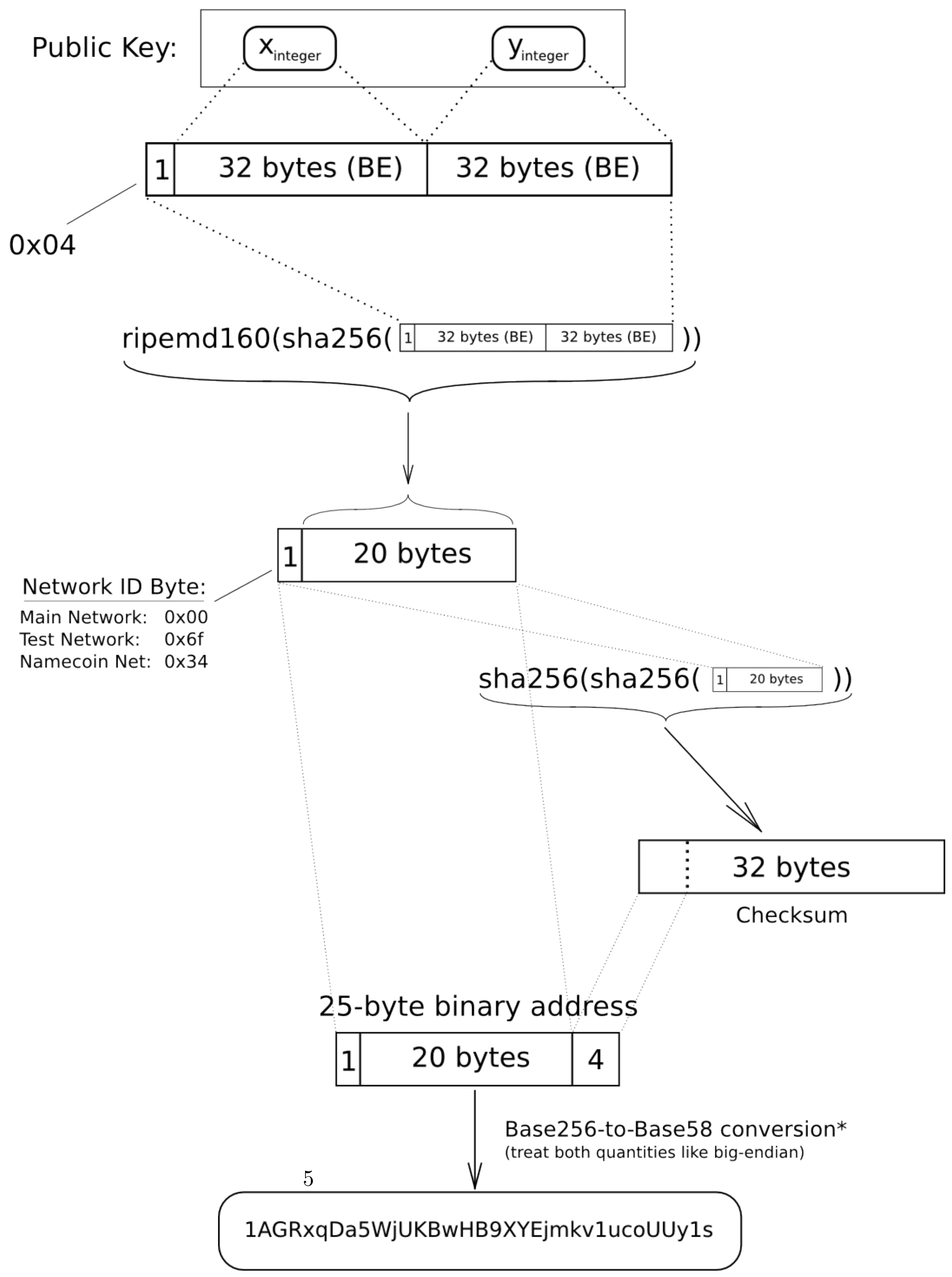
RIPEMD-160 is a cryptographic hash function based upon the Merkle–Damgård construction. It is used in the Bitcoin standard in order to create a Bitcoin address. It is a a strengthened version of the RIPEMD algorithm which produces a 128 bit hash digest while the RIPEMD-160 algorithm produces a 160-bit output. The compression function is made up of 80 stages made up of 5 blocks that run 16 times each. This pattern runs twice with the results being combined at the bottom using modulo 32 addition.
The compression function is made up of a variable sub-block that the message block is passed though 16 times. There are 5 different variations for a total of 80 runs. This process occurs twice with the data meeting at the bottom to be moved on to the next block (if there is one) or added to the hash register is there isn't. The design of the sub-block and the overall layout of the compression function is shown below.

RIPEMD-160 sub-block:



Also elliptic curves are used to build a Bitcoin address.

Elliptic-Curve Public Key to BTC Address conversion



*In a standard base conversion, the 0x00 byte on the left would be irrelevant (like writing '052' instead of just '52'), but in the BTC network the left-most zero chars are carried through the conversion. So for every 0x00 byte on the left end of the binary address, we will attach one '1' character to the Base58 address. This is why main-network addresses all start with '1'

7.2.2 Hash via permutation: Keccak Sponge and SHA3

History:
In the past, SHA1 was used for cryptographic purposes but now it is deprecated since it is not secure: now it is used to compare files only (e.g. in GitHub). SHA2 family is still widely used but then, NIST organization chose SHA3 by means of a competition. A 1600 bits version of the Keccak algorithm was the winner of that competition. Examples of algorithms belonging to SHA3 family are:

- SHA-3-512
- SHA-3-384
- SHA-3-256
- SHA-3-224
- etc.

where the last number is the digest bit length (not the security level!).
Note: in SHA3 family there are also two extendable-output functions (XOFs) named SHAKE128 and SHAKE256 that are used to create a digest of a message of any desired length (in this case the number "128" or "256" indicates the security level (not the digest length). Example: SHAKE128$(msg, 333)$ outputs a 333 bits digest of the message msg.

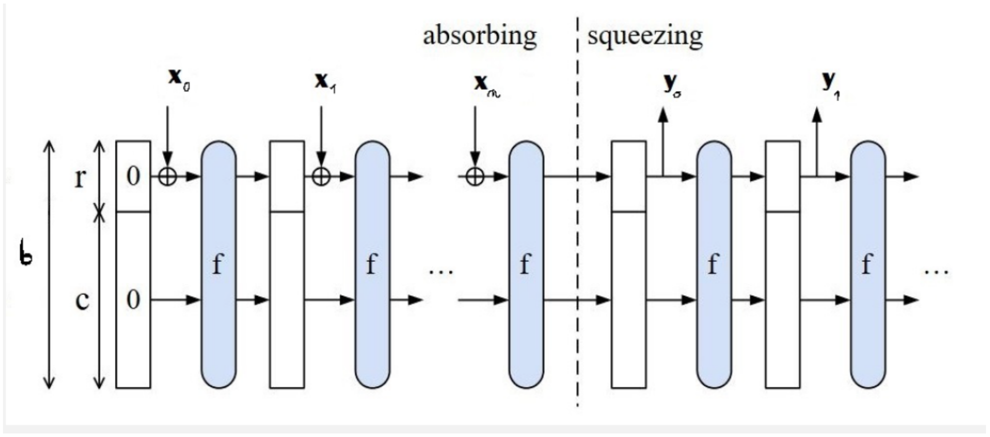
SHA3 (Keccak) algorithm is based on a permutation function f:

$$f : \mathbb{Z}_2^n \to \mathbb{Z}_2^n,\ f \text{ biyective}$$

f should behave as a "Random Oracle": given an input register x, the function f should generate n random bits (computing $f(x)$ should be computationally feasible) and, if the same operation is repeated using the same input x, the result should be the same.

A sponge function or **sponge construction** is any of a class of algorithms with finite internal state that take an input bit stream of any length and produce an output bit stream of any desired length. A random sponge function is a sponge construction where f is a random permutation, for example, Keccak cryptographic sponge with a 1600-bit state (f could be any kind of permutation function).
SHA-3 uses the sponge construction, in which data is "absorbed" into the sponge, then the result is "squeezed" out. In the absorbing phase, message blocks are XORed into a subset of the state, which is then transformed as a whole using a permutation function f. In the "squeeze" phase, output blocks are read from the same subset of the state, alternated with the state transformation function f. The size of the part of the state that is written and read

is called the "rate" (denoted r), and the size of the part that is untouched by input/output is called the "capacity" (denoted c). The capacity determines the security of the scheme. The maximum security level is half the capacity ($\frac{c}{2}$) since it requires a loop of $2^{\frac{c}{2}}$ iterations.



Input x is divided in blocks x_i of r bits (a padding is included). The number b is called "width", sum of the capacity and the bitrate. The state of the sponge is the content of a register of b bits, while f is the permutation of the state, i.e. a $\mathbb{Z}_2^b$.
The two phases are:

1. absorbing: the input r-bits of block x_i are XORed with the r bits of the state. Then f is used to permute the state. This is done until all blocks are "absorbed".

2. squeezing: the output is computed as $y = y_0||y_1||\cdots$ by using the r bits y_i of the state.

The last c bits of the state are not altered by the blocks x_i and they are NOT part of the output.

In SHA-3 Keccak permutation function f, each round consists of a sequence of five steps denoted by Greek letters: θ (theta), ρ (rho), π (pi), χ (chi) and ι (iota). Each step manipulates the entire state.

Keccak permutation function f is invertible so, if the entire register is output, then the procedure will be reversible: this is why only part of it is taken as output at each squeezing step. The capacity determines the claimed level of security, and one can trade claimed security for speed by increasing the capacity c and decreasing the bitrate r accordingly, or viceversa.

- Example (SHA3):
  $b = 1600$, so r could be: 1152, 1088, 832, 576, ...

Comparison between generic Keccak algorithm and SHA3:

- Keccak (also called Keccak-f[b]):
  - b can be freely chosen.
  - Padding is computed as M||10*1 ('1' and '0' are defined bits, while '*' indicates that there is a variable string of 0 bits that depends on value of r).
  - Once b is chosen, then the number of rounds used for computation is known (example: $b = 25 \Rightarrow 12$ rounds [see Section 2 for more details: there's the construction of a keystream by means of Keccak, not of an hash, but the function is the same], since $b = 25 \cdot 2^l$ and so there will be $12 + 2l$ rounds (12 rounds for l=0).
- SHA3:
  - b is fixed (b=1600).
  - Padding is different: M||0110*1
  - In this case there are 24 rounds for $b = 1600$ and $l = 6$: in fact $12 + 2 \cdot 6 = 24$.

Exercise 7.2.6:
The permutation f is not required to be one-way. So why hashing with a sponge should be preimage resistant? Assume that r = 512, b = 1600 and that after padding your message M is x_0. So the hash digest is y_0. Write a loop to find a second preimage of y_0. How many cycles are expected?
Premise: *A public function H from set A to finite set B is:*

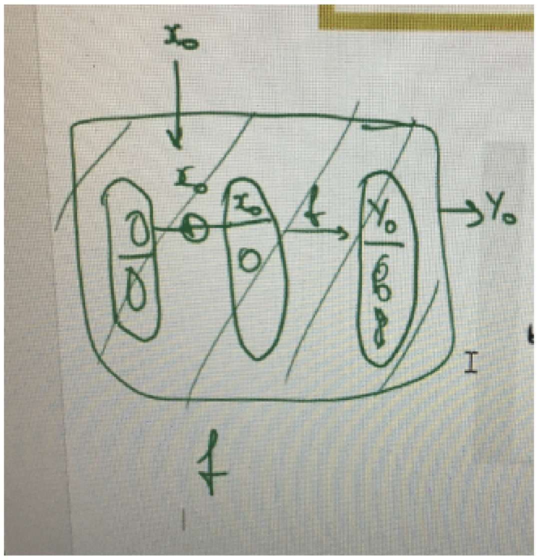
- *first-preimage-resistant when for a given random b in B, it is hard to exhibit a preimage of b, that is, a in A with $H(a) = b$.*
- *Second-preimage-resistant when for a given random a_0 in A, it is hard to exhibit another preimage of $b = H(a_0)$, that is, a in A with $a \neq a_0$ and $H(a) = H(a_0)$ (in other words: Given a* ***fixed*** *input m_1 it should be difficult to find another input m_2 such that $m_1 \neq m_2$ and hash(m_1)=hash(m_2). Functions that lack this property are vulnerable to second-preimage attacks)*
- *Collision-resistant: like second-preimage-resistant but m_1 and m_2 can be both freely chosen by the attacker.*

A preimage can in principle be found by trying various values of a in A (other that a_0 for second-preimage), and computing $H(a)$ until it matches b.

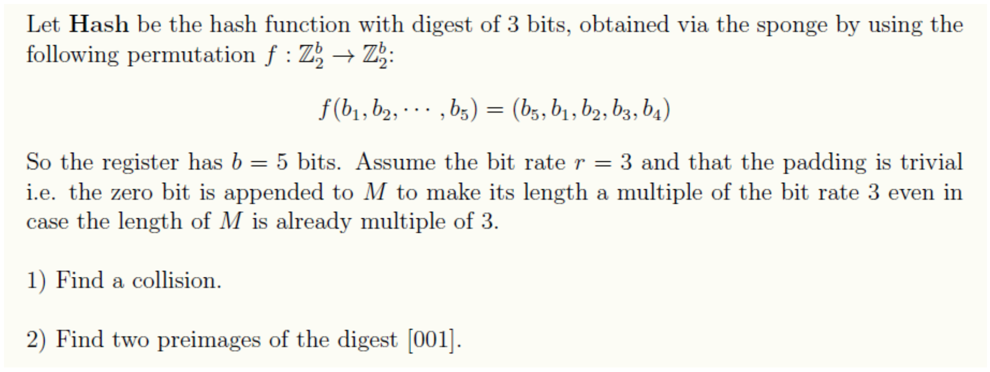
By definition, an ideal hash function is such that the fastest way to compute a first or second preimage is through a brute-force attack. For an n-bit hash, this attack has a time complexity 2^n, which is considered too high for a typical output size of $n = 128$ bits. If such complexity is the best that can be achieved by an adversary, then the hash function is considered preimage-resistant. However, there is a general result that quantum computers perform a structured preimage attack in $\sqrt{2^n} = 2n/2$, which also implies second preimage.

"An iterated function, like this Keccak sponge, uses a finite memory to store its state and processes the input, block per block. At any point in time, the state of the iterated function summarizes the input blocks received so far. Because it contains a finite number of bits, collisions can happen in this state. Random oracles, on the other hand, do not have collisions in their state as such a concept does not exist. This is the main reason for which random oracles cannot be used directly to express security claims of functions with variable-length output: they would simply never exhibit any effects of the finite memory any concrete iterated function has. Random sponges functions, on the other hand, provide an alternative to the random oracle model for expressing security claims. A random sponge is an instance of the sponge construction with f chosen randomly from the set of transformations (or of permutations) over b bits. We have proven that a random sponge function is as strong as a random oracle, except for the effects induced by the finite memory. A random sponge can serve as a reference model for expressing compact security claims for iterated hash functions and stream ciphers." ~ *Keccak team*

Solution: This exercise is about the first step of the sponge **only**. So, to solve this exercise, it is enough to perform a brute force loop: find another x_0 such that the output is again y_0 (the c bits of the final register can be dropped out).



Exercise 7.2.7:



Let **Hash** be the hash function with digest of 3 bits, obtained via the sponge by using the following permutation $f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$:

$$f(b_1, b_2, \cdots, b_5) = (b_5, b_1, b_2, b_3, b_4)$$

So the register has $b = 5$ bits. Assume the bit rate $r = 3$ and that the padding is trivial i.e. the zero bit is appended to M to make its length a multiple of the bit rate 3 even in case the length of M is already multiple of 3.

1) Find a collision.

2) Find two preimages of the digest [001].

1) See point '2)'.

2) In order to obtain [001] as digest and assuming that the input message is only 3 bits long (no padding and no division in blocks are needed) it is necessary to have a register such that the reversed permutation leads to:

$$[01\text{xx}0] \leftarrow [001\text{xx}]$$

So, in order to obtain [01xx0], we need to reverse the XOR operation made with the input x_0:

$$[000\ 00] \oplus [\text{ttt}\ 00] = [01\text{x}\ 00] \rightarrow [01\text{xx}0]$$

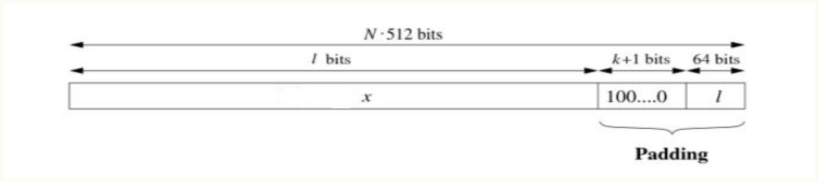
In the end, we have to find the three bits 'ttt', knowing that, XORed with '000' must give the '01x' outcome. There are two possible solutions (the two preimages):
- $[000] \oplus [010] = [010]$
- $[000] \oplus [011] = [011]$

Exercise 7.2.8:

Let **Hash** be the hash function with digest of 512 bit obtained by using the permutation $f : \mathbb{Z}_2^{1024} \to \mathbb{Z}_2^{1024}$ by using the sponge with $b = 1024, r = 512$:

$$f(b_1, b_2, b_3, \cdots, b_{1024}) = (b_{1024}, b_1, b_2, \cdots, b_{1023})$$

Here is the padding:

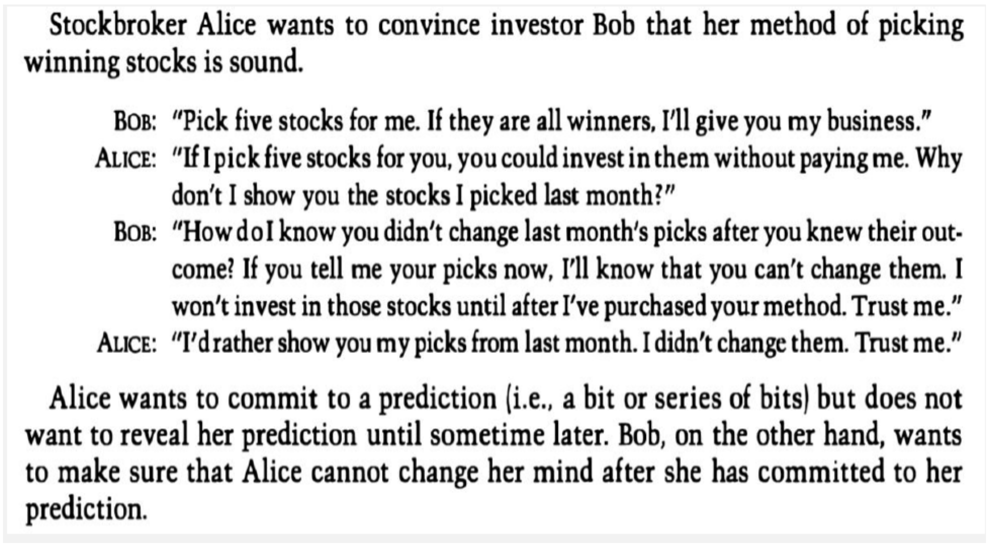


1) Find a collision.

2) Find two preimages of the zero digest $[000 \cdots 0]$.

Solution of this exercise is the same as the previous one, but since the register size is much greater, a coding implementation is needed to perform the operations.

7.3 Bit Commitment



Stockbroker Alice wants to convince investor Bob that her method of picking winning stocks is sound.

BOB: "Pick five stocks for me. If they are all winners, I'll give you my business."
ALICE: "If I pick five stocks for you, you could invest in them without paying me. Why don't I show you the stocks I picked last month?"
BOB: "How do I know you didn't change last month's picks after you knew their outcome? If you tell me your picks now, I'll know that you can't change them. I won't invest in those stocks until after I've purchased your method. Trust me."
ALICE: "I'd rather show you my picks from last month. I didn't change them. Trust me."

Alice wants to commit to a prediction (i.e., a bit or series of bits) but does not want to reveal her prediction until sometime later. Bob, on the other hand, wants to make sure that Alice cannot change her mind after she has committed to her prediction.

A commitment scheme is a protocol for Alice and Bob with two different phases and algorithm Com (Warning: here it is NOT Com="Compression", it is Com="Commitment").

- Commitment phase: Alice has a secret b to commit and sends to Bob a commitment value c (commitment for b). The value c is the output $c = Com(b, r)$ of the algorithm Com; r indicates that c is computed by b and some random value r.
- Opening/Reveal phase: Alice sends to Bob the former secret b so Bob can check if $c' = Com(b, r)$ for r is equal to c (r is pre-shared among Alice and Bob, so Bob knows r).

A commitment scheme is secure if these two properties are satisfied:

1. Hiding property: at the end of the first phase, Bob (even dishonest) does not has any information about b.
2. Binding property: for a given c, there is an unique value of b that convince Bob, so Alice (even dishonest) cannot choose another b.

An application of this scheme is shown in the next exercise:

Exercise 7.3.1: Cryptographic coin flipping
By using a commitment scheme for a single bit ($b \in \{0, 1\}$) construct a protocol for the problem of "flipping a coin by telephone". That is to say, Alice and Bob want to flip a coin by telephone. (They have just divorced, live in different cities, want to decide who gets the car.) Bob would not like to tell Alice HEADS and hear Alice (at the other end of the line) saying "Here goes...I'm flipping the coin... You lost !" Hint: start with some naive protocol e.g. Alice chooses a random bit b_A and send it to Bob. Bob does the same and chooses a random bit b_B and send it to Alice. The output bit for both is $b = b_A \oplus b_B$.

Alice and Bob can use commitments in a procedure that will allow both to trust the outcome:

- Alice "calls" the coin flip but only tells Bob a commitment to her call ($c_A = Com(b_A, r_A)$, c_A sent to Bob)
- Bob flips the coin and reports the result ($c_B = Com(b_B, r_B)$, c_B sent to Alice)
- Alice reveals what she committed to (b_A and r_A are sent to Bob)
- Bob verifies that Alice's call matches her commitment ($c' = Com(b_A, r_A) \rightarrow c' == c_A$?)
- If Alice's revelation matches the coin result Bob reported, Alice wins

For Bob to be able to skew the results to his favor, he must be able to understand the call hidden in Alice's commitment. If the commitment scheme is a good one, Bob cannot skew the results. Similarly, Alice cannot affect the result if she cannot change the value she commits to.

Exercise 7.3.2:
Construct a commitment algorithm Com by using a Hash random oracle.

Bit-commitment schemes are trivial to construct in the random oracle model. Given a hash function H with a $3k$ bit output, to commit the k-bit message m, Alice generates a random k bit string R and sends Bob $H(R||m)$. The probability that any R', m' exist where $m' \neq m$ such that $H(R'||m') = H(R||m)$ is $\approx 2k$, but to test any guess at the message m Bob will need to make $2k$ (for an incorrect guess) or $2k-1$ (on average, for a correct guess) queries to the random oracle. We note that earlier schemes based on hash functions, essentially can be thought of schemes based on idealization of these hash functions as random oracle.

7.4 MAC: Message Authentication Code

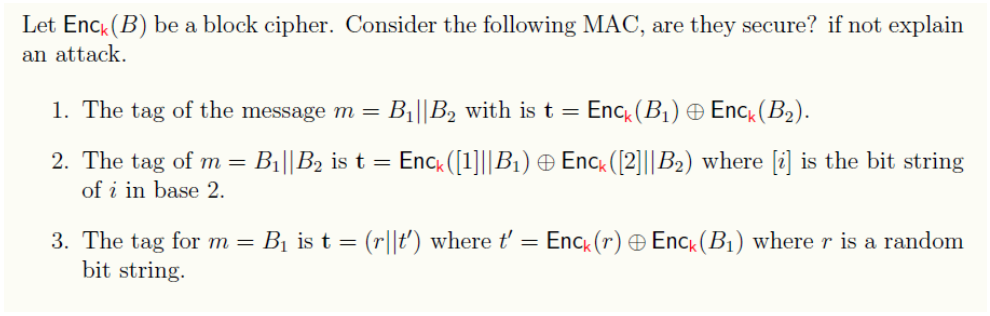
A MAC is a symmetric algorithm that allows to check the authenticity of the message.

A MAC consists of three algorithms:

- $Gen(n)$: the input n is a security parameter and the output is a key k of n bits (*key-generation*). To perform this phase the starting point could be an hash but you must be careful since it could lead to an attack (see later: subsection 7.4.1).
- $Mac_k(m)$: inputs are a message m and a key k. The output is a tag t (*tag-generation*).
- $\text{Vrfy}_k(m)$: inputs are a message m, a key k and a tag t. Output is 1 for "valid tag" or 0 for "invalid tag" (*verification*).

A MAC is secure if an adversary who does not know k, but knows some valid tags $(m_1, t_1) \cdots (m_l, t_l)$, it is not able to produce a valid pair (m, t) where t is a valid tag for the message m, with of course $m \neq m_i$, $\forall i$.

Exercise 7.4.2:



Let $\mathsf{Enc}_k(B)$ be a block cipher. Consider the following MAC, are they secure? if not explain an attack.

1. The tag of the message $m = B_1||B_2$ with is $\mathbf{t} = \mathsf{Enc}_k(B_1) \oplus \mathsf{Enc}_k(B_2)$.
2. The tag of $m = B_1||B_2$ is $\mathbf{t} = \mathsf{Enc}_k([1]||B_1) \oplus \mathsf{Enc}_k([2]||B_2)$ where $[i]$ is the bit string of i in base 2.
3. The tag for $m = B_1$ is $\mathbf{t} = (r||t')$ where $t' = \mathsf{Enc}_k(r) \oplus \mathsf{Enc}_k(B_1)$ where r is a random bit string.

1. In this case, all the messages like $B_i||B_i$ have the same tag $t = [0]$. So, an adversary already knows a lot of combinations.
2. In this case an adversary in CPA scenario can ask to oracle tag of these messages: $B_1||B_2$, $B_1'||B_2andB_1||B_2', sobydoingtheXORitwillobtain$:

$$\cancel{E_k([1]||B_1)} \oplus \cancel{E_k([2]||B_2)} \oplus E_k([1]||B_1') \oplus \cancel{E_k([2]||B_2)} \oplus \cancel{E_k([1]||B_1)} \oplus E_k([2]||B_2')$$

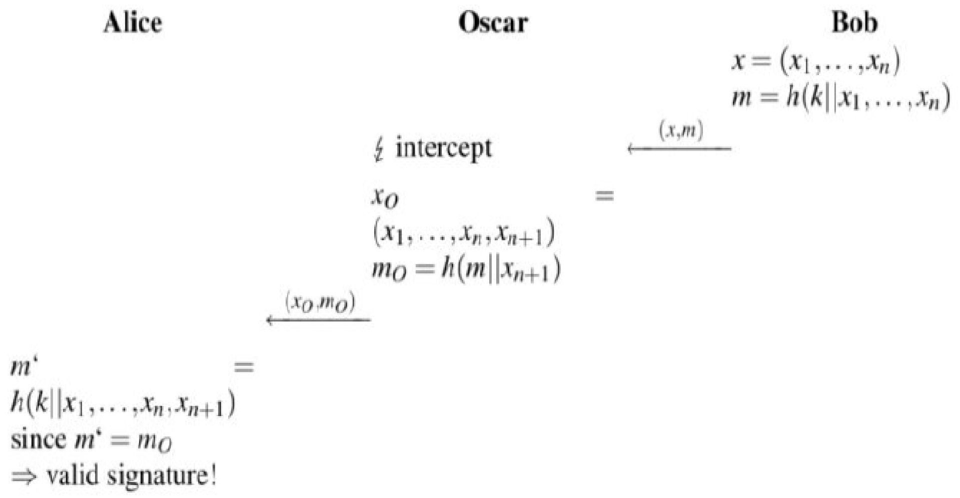
   So it has obtained the tag of message $B_1'||B_2'$,, different from previous three.
3. In this case, if PNRG is secure, also MAC is secure.

7.4.1 Attack: Length extensions & HMAC

This is an attack to the MAC constructed from the hash, not directly to the hash.

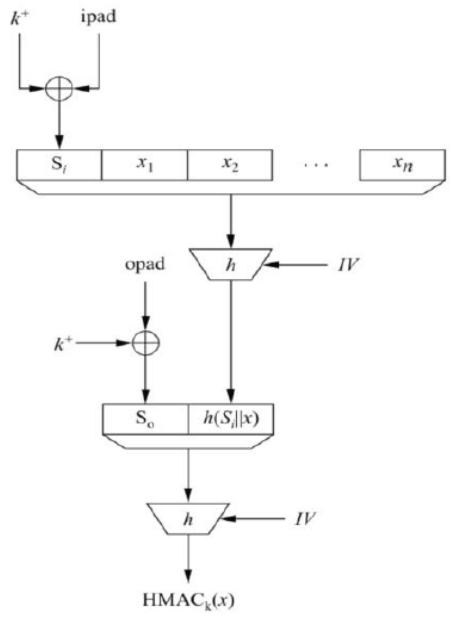
A naive way to get a MAC is by using a hash function: given k and P a tag for the message P is just $t = Hash(k||P)$.

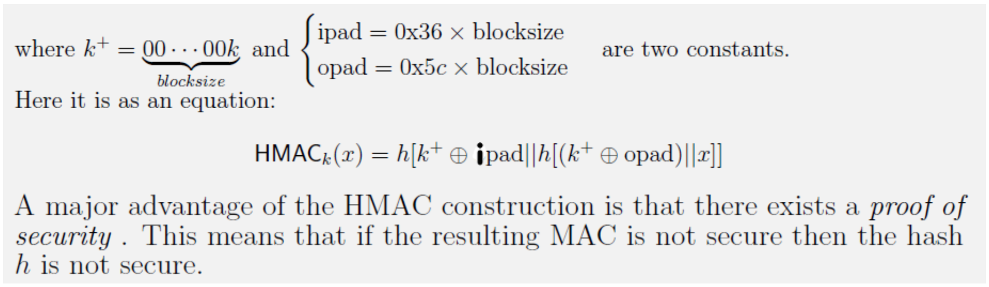
The attack consists in:


Alice
Oscar
Bob
$x = (x_1, \ldots, x_n)$
$m = h(k||x_1, \ldots, x_n)$
(x, m)
↯ intercept
$x_O =$
$(x_1, \ldots, x_n, x_{n+1})$
$m_O = h(m||x_{n+1})$
(x_O, m_O)
$m' =$
$h(k||x_1, \ldots, x_n, x_{n+1})$
since $m' = m_O$
$\Rightarrow$ valid signature!

So the above naive MAC it is not secure if the Hash is designed with the *Merkle-Damgärd* method. This is a special case of a more general Lenght extension attack. This design problem of a Hash is fixed by using HMAC.

HMAC:
A Hash-based Message Authentication Code which does not show the security weakness described above is the HMAC construction, proposed in 1996. The scheme consists of an inner and outer hash.





where $k^+ = \underbrace{00\cdots00k}_{blocksize}$ and $\begin{cases} \text{ipad} = \text{0x36} \times \text{blocksize} \\ \text{opad} = \text{0x5}c \times \text{blocksize} \end{cases}$ are two constants.
Here it is as an equation:

$$\mathsf{HMAC}_k(x) = h[k^+ \oplus \text{ipad}||h[(k^+ \oplus \text{opad})||x]]$$

A major advantage of the HMAC construction is that there exists a *proof of security* . This means that if the resulting MAC is not secure then the hash h is not secure.