Chapter 5. Hashing and Message Authentication[©]

In this chapter, we discuss several efficient cryptographic functions that deal with different aspects of authentication, integrity and commitment to values. We begin with cryptographic hash functions, which provide integrity and commitment solutions, mostly without using any cryptographic keys. We then discuss message authentication codes (MAC), which use a shared secret key to provide message authentication and integrity; MAC functions are often applications of hash functions. In the next chapter, dealing with public key cryptography, we will discuss digital signatures, which provide non-repudiation as well as integrity, authentication and commitment.

5.1. Cryptographic Hash Functions

Let us begin by describing a problem. Alice and Bob want to play the children game of `stone, paper and scissors`, by communicating via an Internet Chat program. Bob suggests kindly that Alice will begin (`Ladies first…`), as in Figure 5.1 below. But Alice knows that if she discloses her choice (paper, stone or scissors), Bob may be tempted to pick the corresponding winning choice (scissors, paper or stone). Alice would like to respond with a hidden commitment to her choice, which does not reveal it; then Bob will do the same, and only then they will disclose the guesses. This `hidden commitment` is illustrated by the function *h()*in Figure 5.1 below, and can be implemented by an appropriate *cryptographic hash function*.

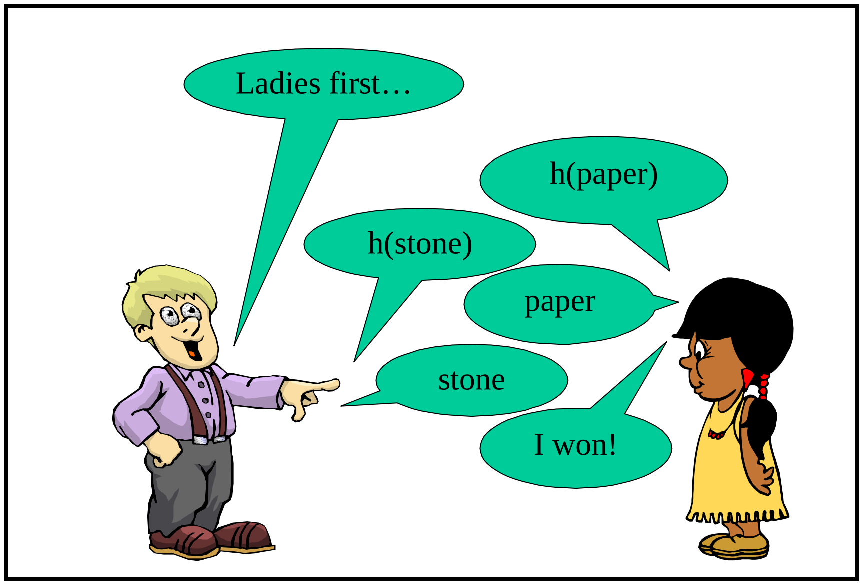

Figure 5.1: Use hash functions to secure `Stone, paper and scissors` game

[©] **COPYRIGHT NOTICE. Copyright © 2001 by author (Amir Herzberg). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission from the author.**

C*ryptographic hash functions*[1] are efficient functions that map a variable-length input string to a fixed-length output string (often 128 or 160 bit), preserving some security property. One has to be careful that the hash function will have the security property required for the specific use, as it is sometimes hard to identify the exact set of security properties which a specific application requires. Practical proposals for cryptographic hash functions, such as MD5 and SHA-1, should be designed to all (or most) of these properties. Still, it is always preferable to require only the weakest properties possible from the hash function, as these properties are usually verified only by cryptoanalysis, and the amount of cryptoanalysis work done for each property is different (and not necessarily always sufficient). The relevant properties and definitions deal with hiding the pre-image, collision resistance, and randomness of the output. We discuss each of these categories in the following subsections.

5.1.1. Hash properties: hiding the pre-image

The most elementary requirement from a cryptographic hash function is that it will be a *one-way function*. We call functions simply *One-Way Hash Function (OWHF);* some other works refer to this property as *pre-image resistance.*

Definition 1 We say that function *f()* is a *one-way function* if it is efficiently computable, yet given the value *f(x)* for input *x* selected randomly from a large uniform distribution of inputs to *f(),* it is infeasible to find the value of *x* or any other value *y* s.t. *f(y)=f(x).* If the domain of *f*() is unbounded and its range is $\{0,1\}^n$ then *f()* is a *One-Way Hash Function (OWHF).*

For example, in the game of Figure 5.1, when Alice first sends the hash of her choice, *h(paper),* if Bob can detect that Alice has chosen *paper*, then Bob may pick scissors and win unfairly. Therefore, we need the hash function to be one-way.

One-way functions are one of the weakest cryptographic primitive functions, and there is substantial amount of research showing how to construct different more complex cryptographic functions, such as pseudo-random generators and block ciphers, from one-way functions.

There are some problems where one-way (hash) functions are sufficient; see for example Exercise 1. However, one-way (hash) functions are often assumed implicitly, or required explicitly, to have much stronger properties than implied by the definition above, for example that the hash value gives no information about the pre-image. This is clearly *not* stated in their definition above; a one-way (hash) function can, e.g., expose some bits in the pre-image, as long as this does not allow guessing the entire pre-image with significant probability. In other cases, the inputs to the hash functions are from a very non-uniform probability distribution (e.g. as in Figure 5.1), in which case it may be possible to find the pre-image (e.g. by computing the hashing of the three values, \`stone\`, \`paper\` and

[1] Other terms for cryptographic hash functions include \`Message Digest\`, \`Fingerprint\`, \`Message Integrity Code (MIC)\` and \`Modification Detection Code (MDC)\`.

`scissors`). Many systems and protocols that use hashing require a stronger property of hiding the inputs.

Following the prudence principle, we may require the hash function to hide everything about the pre-image – make the pre-image indistinguishable from random sample (of the pre-image probability distribution), similar to the definition used for cryptosystems. It is easy to see, however, that no (deterministic) hash function can make the pre-image indistinguishable – we can always distinguish the pre-image by computing its hash value. Therefore, we must either use a weaker definition, or allow the hash process to be randomized, i.e. an additional, random input to the hash function. Canetti suggested in [C97] that it is possible to hide all inputs in a provably secure way, by using a special hash function called oracle hash function. *Oracle hash functions* have two inputs – the regular hash input which we call plaintext, and an additional, random input. Their additional property is that their output does not expose anything about their plaintext input.

Another solution is to `assume the problem away`, and simply assume that the hash function hides everything about the pre-image anyway. Specifically, in the *random oracle model* security is analyzed as if instead of the hash function, the system has oracle access to a truly random function. Proving that the system is secure in the random oracle model does not prove security for general attacks. It is sometimes claimed that security in random oracle model shows that the system is secure against any generic attack, i.e. attacks which work against any hash function and do not use any internal property of the hash function. We are not sure that this claim is justified; maybe there are generic attacks which exploit some property of all hash functions, or of large classes of hash functions, where that property does not exist for a random function. Still, we do believe that the random oracle model does capture many of the generic attacks.

Of course, we cannot really assume that the attacker is not using a different attack; therefore we always prefer a proof that is not assuming a specific attack approach (and all of its assumptions are on the attacker's computational capabilities). See in [CGH98] examples that are secure only under the random oracle model (and insecure with any function). The random oracle model is very useful to eliminate some flawed solutions with a relatively easy analysis. However, although it is often tempting to do so, security in random oracle model should not be trusted as sufficient proof of security, especially when a provable secure solution can be found.

We show some applications of OWHF in section 5.1.4 below; see more examples in exercises 1, 2, and 3.

5.1.2. Hash Properties: Collisions Resistance

Another property often required from hash functions is resistance to collisions. A collision, as in Figure 5.2, is a pair of different pre-image strings, $y \neq x$, which map to the same output, i.e. such that $h(y)=h(x)$. Collision resistance, required by some applications, implies that it is hard to find such collisions. For example, in the game of Figure 5.1, suppose Bob finds a collision, i.e. a value z s.t. $z=h(Stone)=h(Paper)$. After Alice exposes her choice, say *"Stone"*, then Bob will chose to expose *"Paper"*, while if Alice chose otherwise, Bob will chose to expose *"Stone"*. By using a collision-resistant hash function, Bob must commit to his choice before Alice exposes her choice, and the game is fair.

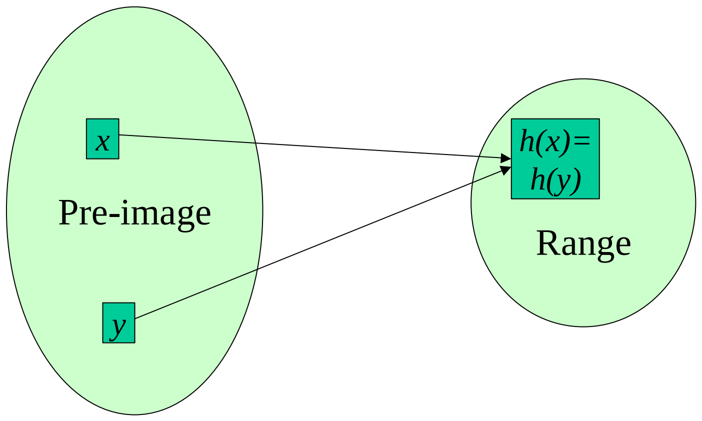

Figure 5.2: A Collision.

The following are common definitions to collision resistance:

- *Weak Collision Resistant Hash Function (weak CRHF)*: a OWHF such that given any input value *x*, it is infeasible to find another input value $y \neq x$ which has the same output, i.e. such that $h(y)=h(x)$. This property is also called *2nd pre-image resistance.*
- *Collision Resistant Hash Function (CRHF):* a OWHF such that it is difficult to find two strings that hash to the same value, i.e. $h(y)=h(x)$. We say that $<x,y>$ is a *collision*. It is easy to see that every CRHF is also a weak CRHF.
- *Correlation (Near Collision) Resistant Hash Function:* a OWHF such that it is hard to find two inputs *x,y* that their hash values differ only in a small number of bits. The precise definition is in [O92]; it essentially says that an attacker cannot do much better than randomly sample pre-image values in attempt to find two which differ in few bits. Anderson [A93] shows that there are some CRHF which are not correlation resistant.
- *Relation Resistant Hash Function:* The correlation-resistance property can be modified to exclude other relations between hash values, e.g. that it will be hard to find *x,y* such that *h(x)* is the complement of *h(y)*. Even more complex relations can be defined (e.g. when considering relations among three inputs, *x,y* and *z*). In general, we say that *h(x)* is *Relation Resistant Hash Function*, where *Relation* is some predicate on $\{0,1\}^{ln}$, if it is hard to find $<x_1,x_2,\ldots x_l>$ s.t. $Relation(h(x_1), h(x_2),\ldots h(x_l))=True$. See motivation and definitions for some important relations, e.g. sum $(h(x)+h(y)=h(z))$, in [A93].

A weak CRHF is sufficient for the very common use of hash functions for non-repudiation. Namely, Alice makes some statement to Bob. For non-repudiation, Alice makes public record of the hash of the statement (for efficiency or privacy). Bob can always present the full statement if there is a dispute, and it will be easy to confirm it using the hash. Even if Alice can select two statements in advance, it will not help it: Bob's statement will be preferred.

Exercise 2 shows that the game of Figure 5.1 requires (at least) a CRHF, not a weakly CRHF; and Exercise 3 demonstrates a scenario where a weakly CRHF is sufficient. For examples where correlation and relation resistance is essential, see [A93].

5.1.3. Hash Properties: Random-looking Output

In the discussion so far (and later on) we try to be careful to use the term `cryptographic hash function` or an even more specific term. This is since hash functions are also used for non-cryptographic purposes, e.g. in data structures and compilation. In these applications, the goal of the hash function is to improve efficiency by using the shortest possible identifiers. Therefore, the outputs of the hash function should not retain any redundancy. Intuitively, we would like them to be uniformly distributed, for any set of (different) inputs.

However, for any fixed, deterministic hash function, the outputs of any fixed set of inputs are also constant. Therefore, this intuitive goal is clearly unfeasible. There may be some weaker goal that can be achieved, but we are not aware of a good definition yet.

Unfortunately, many practical implementations do assume that specific hash functions ensure randomness properties with very weakly random inputs. For example, [RFC1750] suggests generating a short string of random bits by computing a specific, deterministic hash function, in particular MD5, over collection of weakly random inputs. We are not aware that there have been sufficient cryptoanalysis effort of MD5, or other hash functions, to validate such use. Of course, in the random oracle model, such uses are secure.

Add here on universal hashing and compare to other ways to extract randomness e.g. using PRF? Add on Merkle's construction?

5.1.4. Applications of Hash to Identification

5.1.4.1. Unix Password Identification

Traditionally, operating systems used to (and many still does) store the list of passwords `in the clear` in a secure location, accessible only by the password management utilities. This protects the passwords from exposure, as long as the operating system remains secure. However, the password file becomes an obvious target for attackers; exposing it allows an attacker to login as any user of the machine. Unix has a different design, where passwords are never stored `in the clear`. The password file contains only *hash* of the passwords. Unix verifies passwords by computing the hash of the submitted password, and comparing it to the hash in the password file.

The security of this scheme essentially relies on the one-way property of the hash function. Namely, an adversary cannot guess a valid password given its hashed value. However, notice that this holds only if passwords are uniformly distributed. In reality, passwords are often chosen from very biased distribution. This allows a *dictionary attack*, where the adversary computes the results of the hash function over the words in a *dictionary*, which is a list of commonly used passwords (possibly customized for the particular user, e.g. by including variants on personal details such as name, spouse and children names, etc.).

Unix password mechanism contains some defense against dictionary attack. This defense is a 12-bit value, called *salt*, chosen randomly upon password creation. The salt value is used to select the hash function used. Specifically, the hash function used by Unix involves twenty-five repeated applications of a variant of the DES algorithm [DES] over a block of 64 zero bits, with 56 bit of the password as key (7 bits from each of the first 8 characters). The salt is used to modify one of the internal tables of DES. Namely, the password file contains, for each password *p*, the pair $<salt, h_{salt}(p)>$. This hopefully forces the attacker, using a dictionary attack, to compute the hash over each value in the dictionary separately for each user (since each uses a different salt). Furthermore, since h_{salt} uses non-standard DES function, the attacker cannot use standard DES hardware or software to compute the hash.

Some concerns about Unix password security include:

1. Since the passwords are distributed quite non-uniformly, the assumption that the hash function is one-way is not sufficient to ensure that the adversary does not learn information about the password.
2. Since the key length of DES is only 56 bits, this function is subject to exhaustive search.
3. A modified version of DES is used, and non-standard way of constructing a hash from a block cipher. This implies that this usage requires careful analysis (and possible substantial cryptoanalysis efforts) to be considered secure. In spite of the wide use of this technique, very limited work was done in these directions; see some positive results about the general construction, but ignoring the salt-modifications to DES, in [WG00].
4. The passwords are not protected from an eavesdropper listening to the communication between the computer and the terminal, and certainly not from a spoofing program that prompts the user for password while disguising as the system login utility.

We next discuss another identification method based on hashing, which tries to address these concerns.

5.1.4.2. One-time password identification (S/Key)

Lamport [L81] proposed the following simple scheme for identification, which was later deployed in the popular S/Key utility [H94]. The goal of this scheme is to identify a user communicating over an insecure channel, without encryption. In each identification, the user provides the terminal with an identifier, which is then sent over the channel to the server; but each identifier is used only once (for this identification) and cannot be later used by an eavesdropper for imposing as the user. We compute the identifiers by repeated applications of a hash function to the password, and use them in reverse order. Namely, the identifiers are:

$ID_1 = h(password)$
$ID_2 = h(ID_1)$
…
$ID_k = h(ID_{k-1})$

The sequence of hash values computed this way is often referred to as *hash chain.*

Initially, the password file contains only the last identifier in the hash chain, e.g. ID_k. In the first identification, the terminal sends ID_{k-1} to the server. The server maintains the number of identifications and the last identifier received (for the first identification, ID_{k-1}). In subsequent identifications, the terminal exposes one more element in the hash chain each time, until the chain is exhausted (with sending of ID_1 or of *password*). When the chain is exhausted, the user supplies the last identifier of a new chain `in the clear`.

We notice three security weaknesses that are inherent to S/Key's approach:

1. S/Key is not secure against some active attacks (see Exercise 7).
2. If an adversary collects an identifier, she can use this identifier to authenticate herself. Since subsequent messages are not authenticated (S/Key does not establish a shared authentication key), this allows the adversary to gain all the rights of the impersonated user. This weakness is common to all schemes where identification does not involve establishment of message authentication mechanism.
3. Furthermore, the adversary may impose as the user at any time after capturing the identifier, as long as this identifier was not used yet.

The S/Key scheme can be used in several ways:

1. The user may enter his password and the number of identifications so far to the terminal and have the terminal compute the current identifier. This requires relatively `smart` terminal, and complete trust in this terminal. If such a terminal is available, there are alternative protocols that can also establish a shared key to secure the communication, and provide therefore much better security.
2. The user can use a small device to compute the hash values, and manually copy them to the terminal. This allows the terminal to use standard login process designed for regular passwords. S/Key is not the most secure solution for this scenario, since an old identifier may be used.
3. The user uses just a list of one-time passwords on paper. S/Key seems a very good mechanism for this scenario.

The `hash-chain` technique can be applied also to micropayments, as we show in…..

5.1.5. Finding Collision and the Birthday Paradox

5.1.6. Practical Hash Functions and Key Lengths

5.2. Message Authentication Code

One natural use for hash functions is to detect any change in a message. Namely, when transmitting or storing a message, it is sufficient to maintain the output of a CRHF applied to the message, and use it to detect any change to the message. This resembles the use of Error Detection Codes to detect (unintentional) errors in transmitted and stored data. However, we cannot use CRHF for this purpose; the attacker can easily replace the hash value with that of the modified or injected message, as in Figure 5.3. To prevent this attack, the sender must authenticate the message in a way impossible for the adversary.

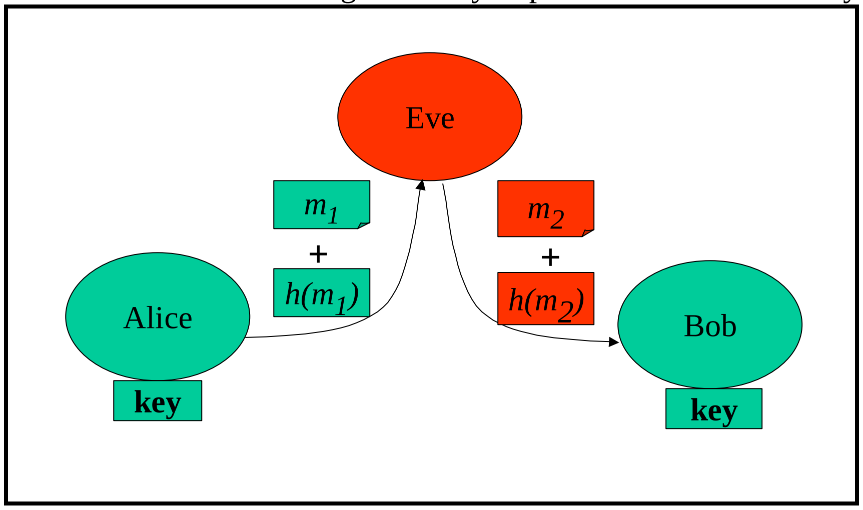

Figure 5.3: Changing Message and its Hash

An efficient solution that prevents an attacker from modifying or injecting messages, is to use a key that is shared between Alice and Bob, but secret from others, and in particular unknown to the adversary (Eve). Alice sends together with the message *m*, another value called the *message authenticator tag* (or simply *tag*) that is the result of a *Message Authentication Code (MAC)*[2] function applied to the message. The MAC function has two inputs – the message to be authenticated *m*, and the secret, shared key. Bob receives the message *m* together with the *tag*, and validates them, again using the secret, shared key. In most MAC designs, Bob validates by applying the MAC function to the message *m* and the key, and comparing the result to the received *tag*. For simplicity, in the rest of the discussion we focus on this validation method (although in principle the validation technique may not require Bob to compute the *MAC*).

[2] Some authors use the term Keyed hash functions to refer to MAC functions. But we find this term confusing (as there are some hash functions which use other sorts of keys, e.g. universals hash functions).

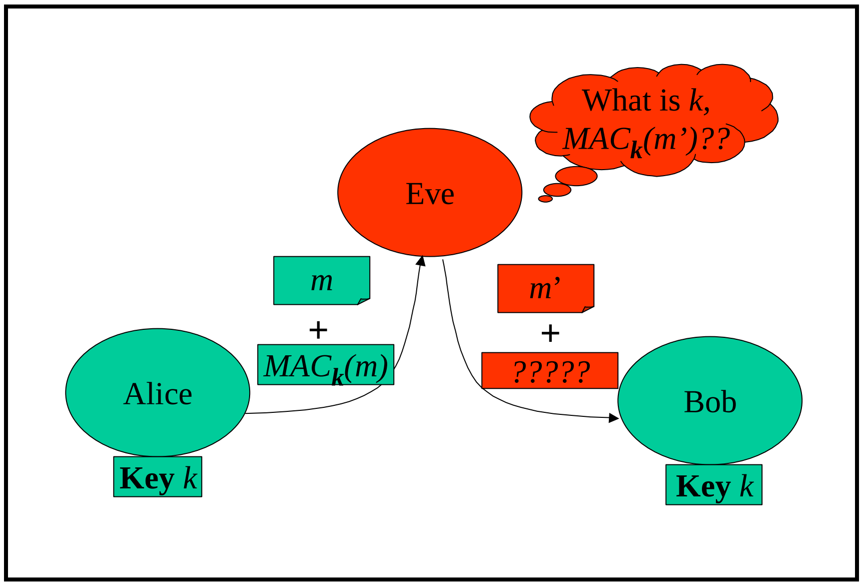

Figure 5.4: Message Authentication Code (MAC)

Definition 2 A (multiplier f or a variable length) *Message Authentication Code* is a function $MAC_k(m)$ with n-bit output and two inputs, an n-bit *key* k and a (nf-bit or variable length, respectively) *message* m, which is secure against Known and/or Chosen message attacks, defined as:

- *Known message attack:* A (small) set of plaintext messages is chosen at random (according to the probability distribution of the input messages). The Adversary receives each message m together with its message authentication code $MAC_k(m)$. Then the adversary needs to produce a new message m', which was not in this set, and an authenticator field for it a', such that with significant probability $a'= MAC_k(m')$.
- *Chosen message attack:* Attacker can repeatedly chose a plaintext message m, and receive the message m together with its message authentication code $MAC_k(m)$. Finally, the adversary needs to produce a new message m' and a' as before, i.e. such that with significant probability $a'= MAC_k(m')$.

Intuitively, it is infeasible to compute $MAC_k(m)$ without access to the key k. Pseudo-random functions from $\{0,1\}^{nf}$ to $\{0,1\}^n$ are a multiplier f MAC functions, and pseudo-random permutations on $\{0,1\}^n$ are multiplier 1 MAC functions.

When used correctly, MAC functions allow two or more mutually trusting parties to authenticate all messages sent between them. Namely:

Claim 1 Chose key k randomly and share it among set T of trusted parties. Assume these parties use key k only to compute $MAC_k(m)$, for different message m. If a party receives m',a' s.t. $a'= MAC_k(m')$, then one of the trusted parties in T previously computed $MAC_k(m')$.

Notice that *k* does not really have to be random – a pseudo-random value is sufficient.

We next discuss practical MAC constructions.

5.2.1. CBC MAC

We now present a MAC function that is a variant of the CBC mode of operation for block ciphers (or pseudo-random permutations), as presented in Section 4.2.5. The MAC variant of CBC mode is part of the [DES] standard, and widely used in practice, especially in banking systems. See Figure 5.5.

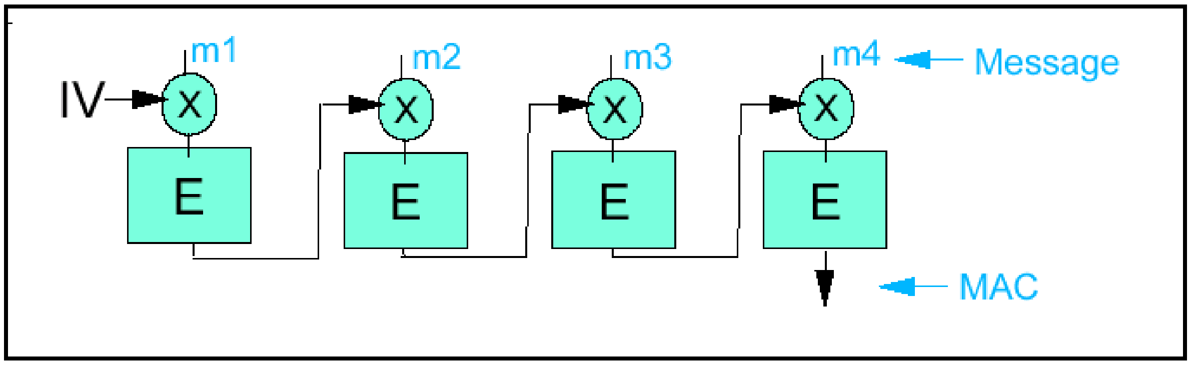

Figure 5.5: The CBC MAC

The CBC MAC function operates on long messages by breaking them into blocks of 64 bits, and applying the block cipher *E* (or PRP) to the bit-wise exclusive-OR of each block and the result of *E* from the previous block. The first block is exclusive-OR with a constant called IV (Initialization Vector), which is not secret or significant for MAC functionality.

In spite of its wide use, the security of the CBC MAC was proved only in 1994 [BKR94]. Also, CBC MAC is secure only when used for fixed-length inputs, namely for a fixed number of blocks. See exercise 9.

5.2.2. HMAC – MAC built from Hash functions

Recently, and especially in Internet protocols, there is more use of MAC built using hashing rather than using block cipher (as CBC MAC). This is motivated by the fact that hash functions are faster than block ciphers in software implementation; software implementations for several hash functions are readily and freely available; and the functions are not subject to export restriction laws and regulations, which sometimes restrict distribution and use of block ciphers. Hashing the message first, and then applying CBC MAC, may address the efficiency motivation; but that adds extra complexity (and still requires the block ciphers to be available).

Some early constructions for MAC based on hashing have only intuitive, heuristic arguments of security. For example, [T92] discussed such constructs as $MAC_k(m)=h(k||m)$, $MAC_k(m)=h(m||k)$ and $MAC_k(m)=h(k||m||k)$, where the symbol || denotes concatenation of strings. All of these are secure in the random oracle model, and [T92] provided some additional heuristics, but there is no proof.

There is no reason to compromise on security here, and indeed, recently most systems adopted a provably secure solution. Bellare, Canetti and Krawczyk presented such a solution in [BCK96]. This solution, the *HMAC* function, is quite simple and efficient, and used in most new practical designs:

$$HMAC_{k}(m)=h(k\oplus opad \,||\, h(k\oplus ipad \,||\, m))$$

Where the values *opad, ipad* are simple constants selected to maximize the hamming distance between $k\oplus opad$ and $k\oplus ipad$. Specifically *opad* is a string of x'36' bytes, and *ipad* is a string of x'5c' bytes. The proof essentially shows that if *h()* is a reasonably secure cryptographic hash function, then HMAC is a secure MAC. Notice that the proof for HMAC still involves a small amount of heuristics. Bellare et al. [BCK96], and later papers, presented also (often similar) designs which were completely provable with no heuristics. However, HMAC was accepted as standard (e.g. by the Internet Engineering Task Force (IETF) [RFC2104]), and seem to be sufficiently secure.

5.2.3. Unconditionally secure MAC?

Add here for completeness?

5.3. Exercises

1. Alice and Bob communicate by phone. Assume they can identify each other's voice, but a hacker, Eve, may eavesdrop on their communication. Alice wants to send a shared key to Bob, carried by Charlie, a completely reliable and trustworthy courier, which is unfortunately not known to Bob. We want Charlie to know some secret so it can prove his identity to Bob by exposing this secret, so that Eve cannot impersonate as Charlie. Show how Alice can establish such a secret using (only) a one-way hash function.
2. Consider the `stone, paper and scissors` game illustrated in Figure 5.1.
   - Demonstrate a problem with this protocol when the hash function is not a CRHF, but only a weakly CRHF.
   - Demonstrate a problem with this protocol with any deterministic hash function.
   - A proposal is made to fix the protocol by sending *h(x,r)* instead of *h(x)*, where *x* is the choice (stone, paper or scissors) and *r* is a random number. Is this secure? Identify assumptions and/or propose another way to make this protocol secure.
3. A person wishes to secure his will from possible modification by a lawyer, by publishing it in the newspaper. For privacy and cost savings, he wants to publish only the hash of the will. Which hash function properties are needed?
4. In the definition of CRHF we required the function to be a OWHF. Demonstrate that this requirement is not redundant (i.e. without it, there could be a CRHF which is not OWHF).
5. [A93] Given a CRHF *h(),* show another CRHF *h'()* which is *not* a correlation resistant hash function. Hint: *h'* can apply *h* to some of its input.
6. A proposal is made to change (improve) Unix security by removing the *salt* value from the password file. On every login, the server will try all possible *salt* values; if there is a match, the login will succeed. Evaluate.
7. Show that S/Key is not secure against active attacks:

- When attacker can intercept the connection where a new chain is initialized.
- When attacker can impersonate as the server.

8. Some designs attempt to provide message authentication by sending the encryption of the message concatenated with its hash (or simply with an error detection code). Namely, they send Encrypt(Message||Hash(Message)), and hope that in so doing, they achieve encryption and authentication together. Show that this design is insecure. Hint: this is easy to show, when using one-time-pad or OFB mode encryption.
9. Find an example showing that CBC MAC is not secure to chosen plaintext attack, when used with variable length input.
10. Often the same message needs to be both signed and encrypted. Show possible weaknesses with the following suggestions.
    1. Send $Sign_{Priv}(Encrypt_k(m))$, providing k for any party that needs to validate the signature.
    2. Send $Sign_{Priv}(m)$, $Encrypt_k(m)$.