Lecture 15: Hashing for Message Authentication

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 22, 2009

©2009 Avinash Kak, Purdue University

Goals:

- What is a hash function?
- Different ways to use hashing for message authentication
- The one-way and collision-resistance properties of secure hash functions
- Simple hashing
- The birthday paradox and the birthday attack
- Structure of cryptographically secure hash functions
- SHA Series of Hash Functions
- Message Authentication Codes

15.1: What is a Hash Function?

- A hash function takes a **variable sized input message** and produces a **fixed-sized output**. The output is usually referred to as the **hash code** or the **hash value** or the **message digest**.

- For example, the SHA-512 hash function takes for input messages of length up to 2^{128} bits and produces as output a 512-bit **message digest** (MD). (**SHA** stands for **Secure Hash Algorithm**. [Note: A series of SHA algorithms has been developed by the National Institute of Standards and Technology and published as Federal Information Processing Standards (FIPS).]

- We can think of the hash code as a **fixed-sized fingerprint** of a variable-sized message.

- Message digests produced by the most commonly used hash functions range in length from 160 to 512 bits depending on the algorithm used.

- Since a message digest depends on all the bits in the input message, any alteration of the input message during transmission would cause its message digest to not match with its original message digest. This can be used to check for forgeries, unauthorized alterations, etc. To see the change in the hash code produced by the most innocuous of changes in the input message:

```
Input message:      "A hungry brown fox jumped over a lazy dog"
SHA1 hash code:     a8e7038cf5042232ce4a2f582640f2aa5caf12d2

Input message:      "A hungry  brown fox jumped over a lazy dog"
SHA1 hash code:     d617ba80a8bc883c1c3870af12a516c4a30f8fda
```

The only difference between the two messages is the extra space between the words "hungry" and "brown" in the second message. Notice how completely different the hash code looks. SHA-1 produces a 160 bit hash code. It takes 40 hex characters to show the code in hex. [The hash codes shown were produced by the following Perl script:

```
#!/usr/bin/perl -w
use Digest::SHA1;
my $hasher =  Digest::SHA1->new();
$hasher->add( "A hungry brown fox jumped over a lazy dog" );
print $hasher->hexdigest;
print ``\n'';
$hasher->add( "A hungry  brown fox jumped over a lazy dog" );
print $hasher->hexdigest;
print ``\n'';
```

As the script shows, this uses the SHA-1 algorithm for creating the message digest. Perl's **Digest** module can be used to invoke any of over fifteen hashing algorithms. The module can output the hash code in either **binary** format, or in **hex** format, or a binary string output as in the form of a **base64**-encoded string. In Python, you can use the **sha** module. Both the **Digest** module for Perl and the **sha** module for Python come with the standard distribution of the languages.]

15.2: Different Ways to Use Hashing for Message Authentication

Figures 1 and 2 show six different ways in which you could incorporate message hashing in a communication network. These constitute different approaches to **protect the hash value** of a message. No authentication at the receiving end could possibly be achieved if both the message and its hash value are accessible to an adversary wanting to tamper with the message. To explain each scheme separately:

- In the symmetric-key encryption based scheme shown in Figure 1(a), the message and its hash code are concatenated together to form a composite message that is then encrypted and placed on the wire. The receiver decrypts the message and separates out its hash code, which is then compared with the hash code calculated from the received message. The hash code provides authentication and the encryption provides confidentiality.

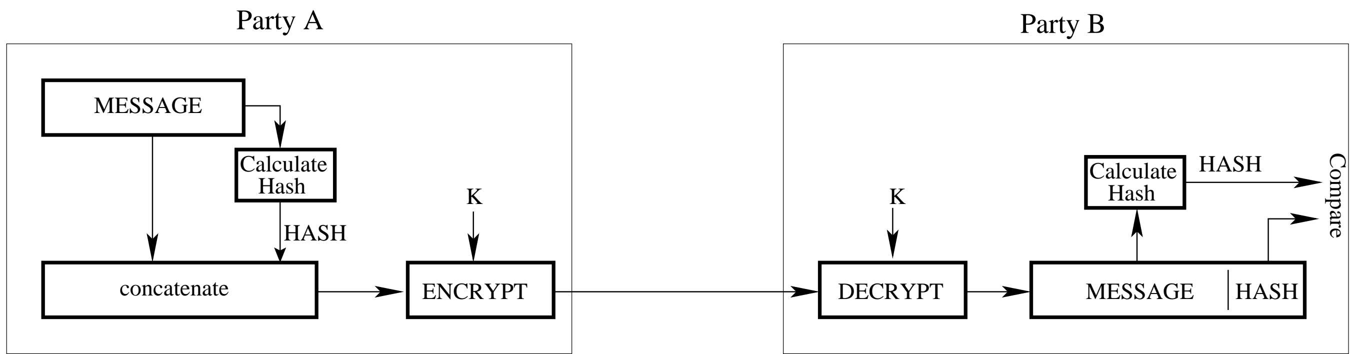
- The scheme shown in Figure 1(b) is a variation on Figure 1(a) in the sense that only the hash code is encrypted. This scheme is efficient to use when confidentiality is not the issue but message authentication is critical. Only the receiver with access to the secret key knows the real hash code for the message. So the receiver can verify whether or not the message is authentic.

- The scheme in Figure 1(c) is a public-key encryption version of the scheme shown in Figure 1(b). The hash code of the message is encrypted with the sender's private key. The receiver can recover the hash code with the sender's public key and authenticate the message as indeed coming from the alleged sender. Confidentiality again is not the issue here. **The sender encrypting with his/her private key the hash code of his/her message constitutes the basic idea of digital signatures.**

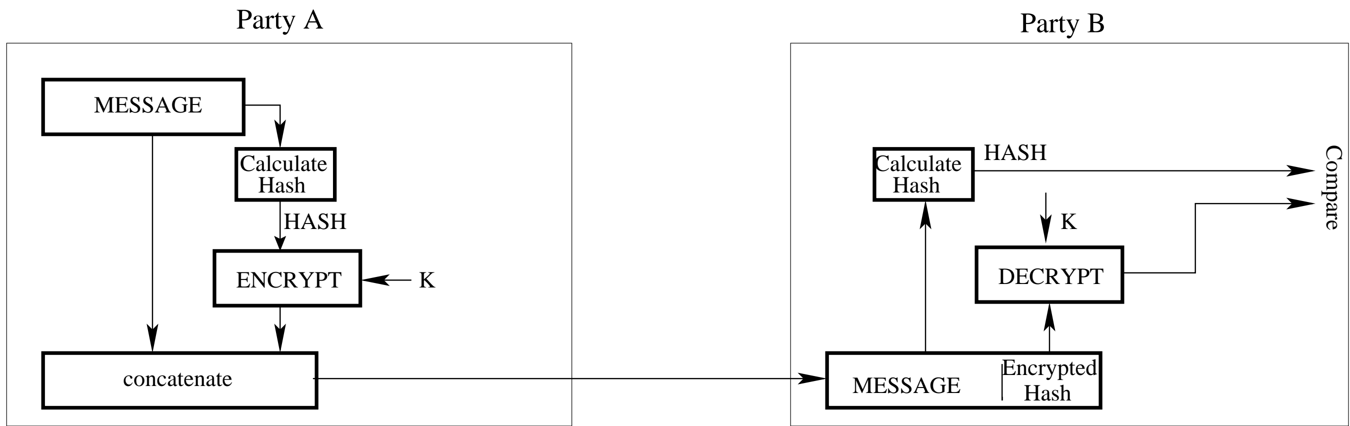
- If we want to add symmetric-key based confidentiality to the scheme of Figure 1(c), we can use the scheme shown in Figure 2(a). This is a commonly used approach when both confidentiality and authentication are needed.

- A very different approach to the use of hashing for authentication is shown in Figure 2(b). In this scheme, nothing is encrypted. However, the sender appends a secret string S, known also to the receiver, to the message before computing its hash code. Before checking the hash code of the received message for its authentication, the receiver appends the same secret string S to the message. Obviously, it would not be possible for anyone to alter such a message, even when they have access to both the original message and the overall hash code.

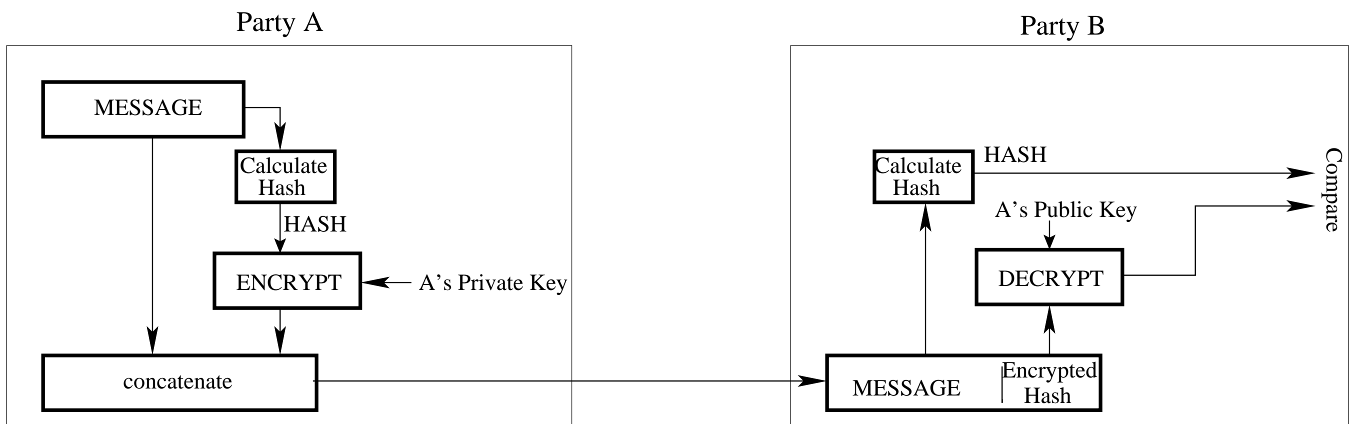
- Finally, the scheme in Figure 2(c) shows an extension of the scheme of Figure 2(b) where we have added symmetric-key based confidentiality to the transmission between the sender and the receiver.



(a)

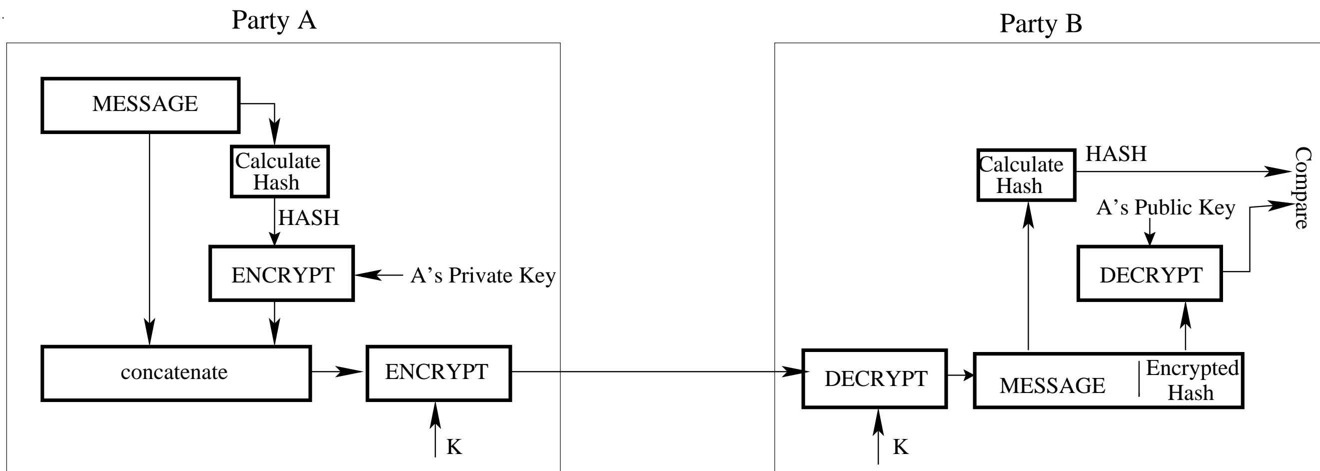


(b)

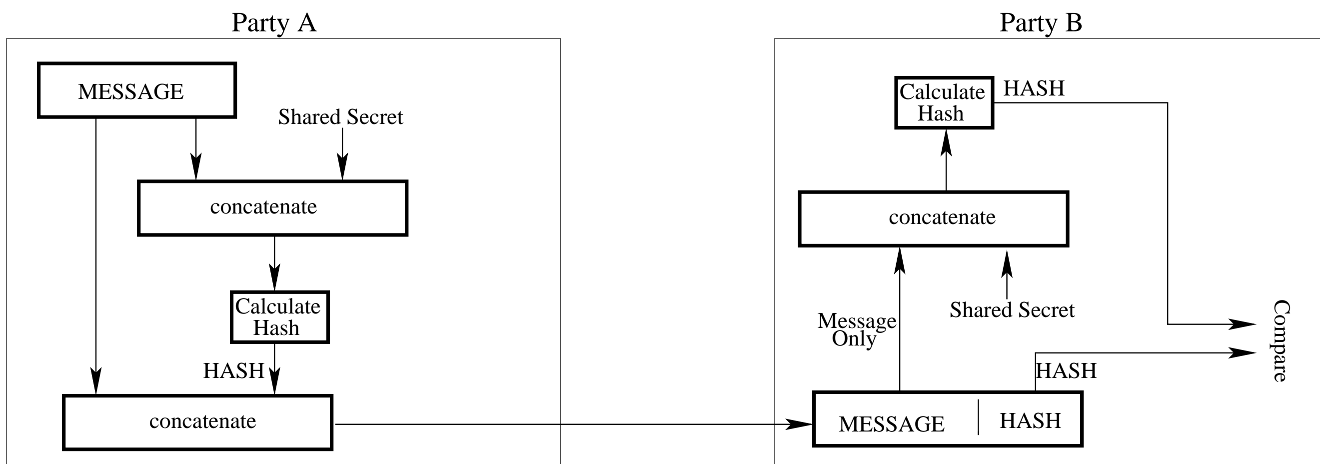


(c)

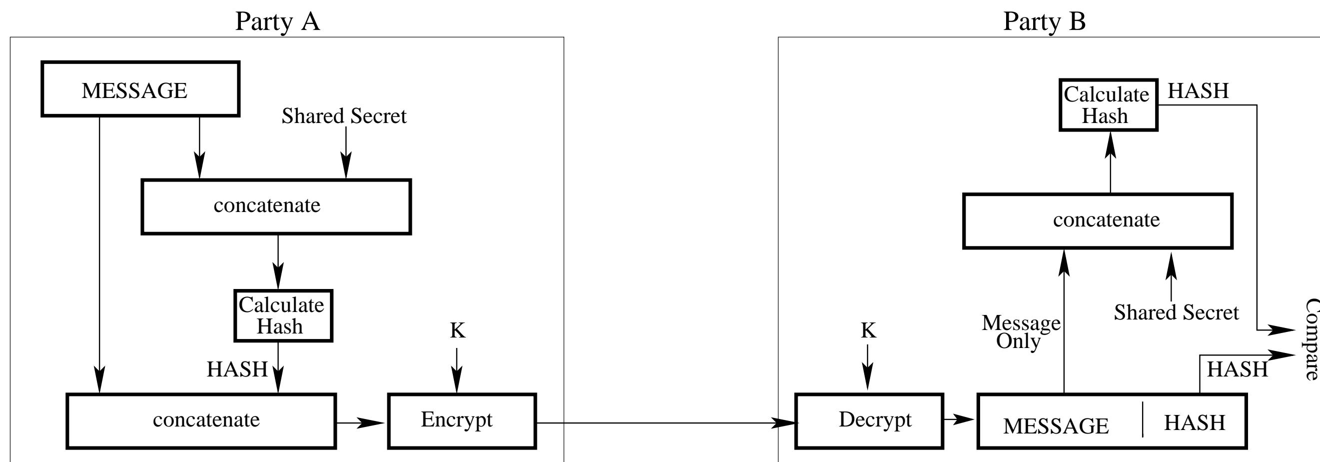
Figure 1: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*



(a)



(b)



(c)

Figure 2: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

15.3: When is a Hash Function Secure?

- A hash function is called **secure** if the following two conditions are satisfied:

  - If it is **computationally infeasible** to find a message that corresponds to a **given** hash code. This is sometimes referred to as the **one-way property** of a hash function.

  - If it is **computationally infeasible** to find two **different messages** that hash to the same hash code value. This is also referred to as the **strong collision resistance** property of a hash function.

- A weaker form of the strong collision resistance property is that for a **given message**, there should not correspond another message with the same hash code.

- Hash functions that are **not collision resistant** can fall prey to **birthday attack**. More on that later.

- If you use n bits to represent the hash code, there are only 2^n **distinct** hash code values. If we place on constraints whatsoever on the messages, then obviously there will exist multiple messages giving rise to the same hash code. But then considering messages with no constraints whatsoever does not represent reality because messages are not noise — they must possess considerable structure in order to be intelligible to humans. **Collision resistance refers to the likelihood that two different messages possessing certain basic structure so as to be meaningful will result in the same hash code.**

- Ideally (if authentication is the only issue and we are not concerned about confidentiality), to ward off message alteration by en-route ill-intentioned agents, we would like to send unencrypted plaintext messages with encrypted hash codes. **(This eliminates the computational overhead of encryption and decryption for the main message content and yet allows for authentication.)** But this only works when collision resistance is perfect. If a hashing approach has poor collision resistance, all that an adversary has to do is to compute the hash code of the message content and replace it with some other content that has the same hash code value. **The fact that the hash code value is encrypted does not do us any good here.**

15.4: Simple Hash Functions

- Practically all algorithms for computing the hash code of a message view the message as a sequence of n-bit blocks.

- The message is processed one block at a time in an iterative fashion to produce an n-bit hash code.

- Perhaps the simplest hash function consists of starting with the first n-bit block, XORing it bit-by-bit with the second n-bit block, XORing the result with the next n-bit block, and so on. We will refer to this as the XOR hash algorithm.

- With this algorithm, every bit of the hash code represents the parity at that bit position if we look across all of the b-bit blocks. For that reason, the hash code produced is also known as **longitudinal parity check**.

- The hash code generated by the XOR algorithm can be useful as a **data integrity check** in the presence of completely random transmission errors. But, in the presence of an adversary trying

to deliberately tamper with the message content, the XOR algorithm is useless for message authentication. **An adversary can modify the main message and add a suitable bit block before the hash code so that the final hash code remains unchanged.**

- Another problem with this simple algorithm is its somewhat reduced collision resistance for structured documents. Ideally, one would hope that, with an n-bit hash code, any particular message would result in a given hash code value with a probability of $\frac{1}{2^n}$. But now consider the case when the characters in a text message are represented by their ASCII codes. Since the highest bit in each byte for each character will always be 0, you can see that some of the n bits in the hash code will predictably be 0 with the simple XOR algorithm. **This obviously reduces the number of unique hash code values available to us, and thus increases the probability of collisions.**

- To increase the space of distinct hash code values available for the different messages, a variation on the basic XOR algorithm consists of performing a one-bit circular shift of the partial hash code obtained after each n-bit block of the message is processed. This algorithm is known as the rotated-XOR algorithm (ROXR).

- That the collision resistance of ROXR is also poor is obvious from the fact that we can take a message M_1 along with its hash code value h_1; replace M_1 by a message M_2 of hash code value h_2; append a block of gibberish at the end M_2 to force the hash code value of the composite to be h_1. So even if M_1 was transmitted with an encrypted h_1, it does not do us much good from the standpoint of authentication. **We will see later how secure hash algorithms make this ploy impossible by including the length of the message in what gets hashed.**

- As a quick example of including the length of the message in what gets hashed, here is how the very popular SHA-1 algorithm pads the message before it is hashed:

```
The very first step in the SHA1 algorithm is to pad the message
so that it is a multiple of 512 bits.

This padding occurs as follows (from NIST FPS 180-2):

  Suppose the length of the message M is L bits.

  Append bit 1 to the end of the message, followed by K
  zero bits where K is the smallest nonnegative solution to

       L + 1 + K  =   448 mod 512

  Next append a 64-bit block that is a binary representation
  of the length integer L.

  Consider the following example:

  Message     =     "abc"
  length L    =     24 bits

  This is what the padded bit pattern would look like:

  01100001  01100010   01100011   1 00......000   00...011000
     a         b        c          <---423--->   <---64---->
  <--------------------   512  ------------------------------->
```

15.5: What does Probability Theory Have to Say about a Randomly Produced Message Having a Particular Hash Code Value?

- Assume that we have **random message generator** and that we calculate the hash code for each message.

- Let's say we have in our possession a message x whose hash code is $h(x)$.

- Let's consider a pool of k messages produced randomly by this generator. Since we are not placing any constraints on messages, there is an infinite number of different messages that the generator can produce. So the probability that any of the k messages in the pool is the same as x is practically 0.

- Now we pose the following question: What is the value of k so that the pool contains **at least one** message y for which the probability of $h(y)$ being equal to $h(x)$ is 0.5?

- To find k, we reason as follows:

– Let's say that the hash code can take on N different values. If the message generator is truly random in its construction of messages, all hash code values will be equally probable.

– Say we pick massage y at random from the pool of messages. The probability that $h(y)$ has any particular value is $\frac{1}{N}$. Since $h(x)$ is given, the probability that $h(y)$ equals $h(x)$ is $\frac{1}{N}$.

– Since $h(y)$ either equals $h(x)$ or does not equal $h(x)$, the probability that $h(y)$ does **not** equal $h(x)$ is $1 - \frac{1}{N}$.

– It follows that the probability that **none** of the messages in a pool of k messages has its hash codes equal to $h(x)$ is $(1-\frac{1}{N})^k$.

– Therefore, the probability that **at least one** of the k messages has its hash code equal to $h(x)$ is

$$1 \; - \; \left(1 - \frac{1}{N}\right)^k \qquad (1)$$

– The probability expression shown above can be considerably simplified by recognizing that as a approaches 0, we can write

$(1+a)^n \approx 1+an$. Therefore, the probability expression we derived can be approximated by

$$\approx \quad 1 \; - \; \left(1 - \frac{k}{N}\right) \quad = \quad \frac{k}{N} \qquad (2)$$

- So the upshot is that, given a pool of k randomly produced messages, the probability there will exist at least one message in this pool whose hash code equals the given value $h(x)$ is $\frac{k}{N}$.

- Let's now go back to the original question: How large should k be so that the pool of messages contains at least one message whose hash code equals the given value $h(x)$ with a probability of 0.5? We obtain the value of k from the equation $\frac{k}{N} = 0.5$. That is, $k = 0.5N$.

- Consider the case when we use 64 bit hash codes. In this case, $N = 2^{64}$. We will have to construct a pool of 2^{63} messages so that the pool contains at least one message whose hash code equals $h(x)$ with a probability of 0.5.

15.6: What is the Probability that a Pair of Messages will Have the Same Hash Code Value?

- Given a pool of k messages, the question "*What is the probability that any message in the pool has its hash code equal to a particular value?*" **is very different from the question** "*What is the probability that any pair of messages in the pool will have the same hash code?*"

- The question "*What is the probability that, in a class of* 20 *students, someone else has the same birthday as yours?*" **is very different from the question** "*What is the probability that there exists at least one pair of students in a class of* 20 *students with the same birthday?*" The probability of the former is approximately $\frac{19}{365}$, and the probability of the latter is roughly the much larger value $\frac{19\times18/2}{365} = \frac{171}{365}$ (This is referred to as the **birthday paradox**, paradox only in the sense that it seems counterintuitive.)

- Given a pool of k messages, each of which has a hash code value from N possible such values, the probability that the pool will contain at least one pair with identical hash code value is given by

$$1 \quad - \quad \frac{N!}{(N-k)!N^k} \tag{3}$$

- The following reasoning establishes the above result. The reasoning consists of figuring out the total number of ways (M_1) in which we can construct a pool of k message with no duplicate hash codes and the total number of ways (M_2) we can do the same while allowing for duplicates. The ratio M_1/M_2 then gives us the probability of constructing a pool of k messages with no duplicates. Subtracting this from 1 yields the probability that the pool of k messages will have at least one duplicate hash code.

  – Let's consider in how many different ways we can construct a pool of k messages so that we are guaranteed to have no duplicate hash codes in the pool. For the sake of this mental experiment, let's assume that we have available to us a very large set of **randomly generated** message – hash-code pairs, that is randomly generated pairs of $\{x, h(x)\}$, where x is a message and $h(x)$ its hash code.

  – For the first message in the pool, we can choose any arbitrarily. Since there are only N different messages with distinct hash codes, so there are N ways to choose the first entry for

the pool. Stated differently, there is a choice of N different candidates for the first entry in the pool.

– Having used up one hash code, we can select a message corresponding to the other $N-1$ still available hash codes for the second entry for the pool.

– Having used up two distinct hash code values, we can select a message corresponding to the other $N-2$ still available hash codes for the third entry for the pool; and so on.

– Therefore, the total number of ways, M_1, in which we can construct a pool of k messages with **no** duplications in hash code values is

$$M_1 \quad = \quad N \times (N-1) \times \ldots \times (N-k+1) \quad = \quad \frac{N!}{(N-k)!} \qquad (4)$$

– Let's now try to figure out the total number of ways, M_2, in which we can construct a pool of k messages without worrying at all about duplicate hash codes. Reasoning as before, there are N ways to choose the first message. For selecting the second message, we pay no attention to the hash code value of the first message. There are still N ways to select the second

message; and so on. Therefore, the total number of ways we can construct a pool of k messages without worrying about hash code duplication is

$$M_2 \quad = \quad N \times N \times \ldots \times N \quad = \quad N^k \tag{5}$$

– Therefore, the *probability* of constructing a pool of k messages with no duplications in hash codes is

$$\frac{M_1}{M_2} \quad = \quad \frac{N!}{(N-k)!N^k} \tag{6}$$

– Therefore, the probability of constructing a pool of k messages with at least one duplication in the hash code values is

$$1 \quad - \quad \frac{N!}{(N-k)!N^k} \tag{7}$$

- The probability expression in Equation (3) (or Equation (7) above) can be simplified by rewriting it in the following form:

$$1 \quad - \quad \frac{N \times (N-1) \times \ldots \times (N-k+1)}{N^k} \tag{8}$$

which is the same as

$$1 \quad - \quad \frac{N}{N} \times \frac{N-1}{N} \times \ldots \times \frac{N-k+1}{N} \tag{9}$$

and that is the same as

$$1 \quad - \quad \left[\left(1 - \frac{1}{N}\right) \times \left(1 - \frac{2}{N}\right) \times \ldots \times \left(1 - \frac{k-1}{N}\right)\right] \tag{10}$$

- We will now use the approximation that $(1 - x) \leq e^{-x}$ for all $x \geq 0$ to make the claim that the above probability is lower-bounded by

$$1 \quad - \quad \left[e^{-\frac{1}{N}} \times e^{-\frac{2}{N}} \times \ldots \times e^{-\frac{k-1}{N}}\right] \tag{11}$$

- Since $1 + 2 + 3 + \ldots + (k - 1)$ is equal to $\frac{k(k-1)}{2}$, we can write the following expression for the lower bound on the probability

$$1 \quad - \quad e^{-\frac{k(k-1)}{2N}} \tag{12}$$

So the probability that a pool of k messages will have at least one pair with identical hash codes is always greater than the value given by the above formula.

- We can use the above formula to estimate the size k of the pool so that the pool contains at least one pair of messages with equal hash codes with a probability of 0.5. We need to solve

$$1 \quad - \quad e^{-\frac{k(k-1)}{2N}} \quad = \quad \frac{1}{2}$$

Simplifying, we get

$$e^{\frac{k(k-1)}{2N}} = 2$$

Therefore,

$$\frac{k(k-1)}{2N} = ln2$$

which gives us

$$k(k-1) = (2ln2)N$$

- Assuming k to be large, the above equation gives us

$$k^2 \approx (2ln2)N \qquad (13)$$

implying

$$\begin{aligned} k &\approx \sqrt{(2ln2)N} \\ &\approx 1.18\sqrt{N} \\ &\approx \sqrt{N} \end{aligned}$$

- So our final result is that if the hash code can take on a total N different values, a pool of $\sqrt{N}$ messages will contain at least one pair of messages with the same hash code with a probability of 0.5.

- So if we use an n-bit hash code, we have $N = 2^n$. In this case, a message pool of $2^{n/2}$ randomly generated messages will contain at least one with a specified value for the hash code with a probability of 0.5.

- Let's again consider the case of 64 bit hash codes. Now $N = 2^{64}$. So a pool of 2^{32} randomly generated messages will have at least one pair with identical hash codes with a probability of 0.5.

15.7: The Birthday Attack

- This attack applies to the following scenario: Say A has a dishonest assistant B preparing contracts for A's digital signature.

- B prepares the legal contract for a transaction. B then proceeds to create a large number of variations of the legal contract without altering the legal content of the contract and computes the hash code for each. These variations may be constructed by mostly innocuous changes such as the insertion of additional white space between some of the words, or contraction of the same; insertion or or deletion of some of the punctuation, slight reformatting of the document, etc.

- B prepares a fraudulent version of the contract. As with the correct version, B prepares a large number of variations of this contract, using the same tactics as with the correct version.

- Now the question is: "*What is the probability that the two sets of contracts will have at least one contract each with the same hash code?*"

- Let the set of variations on the correct form of the contract be denoted $\{c_1, c_2, \ldots, c_k\}$ and the set of variations on the fraudulent contract by $\{f_1, f_2, \ldots, f_k\}$. **We need to figure out the probability that there exists at least one pair** (c_i, f_j) **so that** $h(c_i) = h(f_j)$**.**

- If we assume (**a very questionable assumption indeed**) that all the fraudulent contracts are truly random vis-a-vis the correct versions of the contract, then the probability of f_1's hash code being any one of N permissible values is $\frac{1}{N}$. Therefore, the probability that the hash code $h(c_1)$ matches the hash code $h(f_1)$ is $\frac{1}{N}$. Hence the probability that the hash code $h(c_1)$ does **not** match the hash code $h(f_1)$ is $1 - \frac{1}{N}$.

- Extending the above reasoning to joint events, the probability that $h(c_1)$ does **not** match $h(f_1)$ **and** $h(f_2)$ **and** ..., $h(f_k)$ is

$$\left(1 - \frac{1}{N}\right)^k$$

- The probability that the same holds conjunctively for all members of the set $\{c_1, c_2, \ldots, c_k\}$ would therefore be

$$\left(1 - \frac{1}{N}\right)^{k^2}$$

This is the probability that there will NOT exist any hash code matches between the two sets of contracts $\{c_1, c_2, \ldots, c_k\}$ and $\{f_1, f_2, \ldots, f_k\}$.

- Therefore the probability that there will exist **at least one** match in hash code values between the set of correct contracts and the set of fraudulent contracts is

$$1 \quad - \quad \left(1 - \frac{1}{N}\right)^{k^2}$$

- Since $1 - \frac{1}{N}$ is always less than $e^{-\frac{1}{N}}$, the above probability will always be greater than

$$1 \quad - \quad \left(e^{-\frac{1}{N}}\right)^{k^2}$$

- Now let's pose the question: "*What is the least value of k so that the above probability is 0.5?*" We obtain this value of k by solving

$$1 \quad - \quad e^{-\frac{k^2}{N}} \quad = \quad \frac{1}{2}$$

which simplifies to

$$e^{\frac{k^2}{N}} \quad = \quad 2$$

which gives us

$$k \quad = \quad \sqrt{(\ln 2)N} \quad = \quad 0.83\sqrt{N} \quad \approx \quad \sqrt{N}$$

So if B is willing to generate $\sqrt{N}$ versions of the both the correct contract and the fraudulent contract, there is better than an even chance that B will find a fraudulent version to replace the correct version.

- If n bits are used for the hash code, $N = 2^n$. In this case, $k = 2^{n/2}$.

- The birthday attack consists, as you'd expect, of B getting A to digitally sign a correct version of the contract and then replacing the contract by its fraudulent version that has the same hash code value. The fact that A would encrypt the hash code with his/her private key is of no consequence.

- This attack is called the birthday attack because the combinatorial issues involved are the same as in the birthday paradox presented earlier. Also note that for n-bit hash codes, the value of k the approximate value we obtained for k is the same in both cases, that is $2^{n/2}$.

15.8: Structure of Cryptographically Secure Hash Functions

- A hash function is cryptographically secure if it is computationally infeasible to find collisions, that is if it is computationally infeasible to construct meaningful messages whose hash code would equal a specified value. Said another way, a hash function should be strictly **one-way**, in the sense that it lets us compute the hash code for a message, but does not let us figure out a message for a given hash code.

- Most secure hash functions are based on the structure proposed by Merkle. This structure forms the basis of SHA series of hash functions and also the Whirlpool hash function.

- The input message is partitioned into L bit blocks, each of size b bits. If necessary, the final block is padded suitably so that it is of the same length as others.

- The final block also includes the total length of the message whose hash function is to be computed. This step enhances the security of the hash function since it places an additional constraint on

the counterfeit messages.

- Merkle's structure, shown in Figure 3, consists of L stages of processing, each stage processing one of the b-bit blocks of the input message.

- Each stage of the structure in Figure 3 takes two inputs, the b-bit block of the input message meant for that stage and the n-bit output of the previous stage.

- For the n-bit input, the first stage is supplied with a special N-bit pattern called the **Initialization Vector** (IV).

- The function f that processes the two inputs, one n bits long and the other b bits long, to produce an n bit output is usually called the **compression function**. That is because, usually, $b > n$, so the output of the f function is shorter than the length of the input message segment.

- The function f itself may involve **multiple rounds of processing** of the two inputs to produce an output.

- The precise nature of f depends on what hash algorithm is being implemented, as we will see in the rest of this lecture.

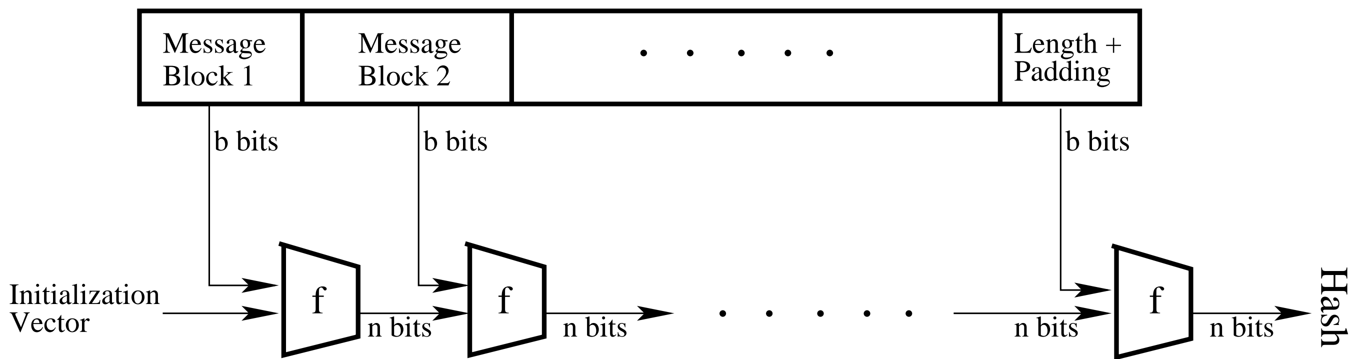


Figure 3: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

15.9: The SHA Family of Hash Functions

- SHA (Secure Hash Algorithm) refers to a family of NIST-approved cryptographic hash functions.

- The most commonly used hash function from the SHA family is SHA-1. It is used in many applications and protocols that require secure and authenticated communications. SHA-1 is used in SSL/TLS, PGP, SSH, S/MIME, and IPSec. (*These standards will be briefly reviewed in Lecture 20.*)

- The following table shows the various parameters of the different SHA hashing functions.

Algorithm	*Message Size* (bits)	*Block Size* (bits)	*Word Size* (bits)	*Message Digest Size* (bits)	*Security* (bits)
SHA-1	$< 2^{64}$	512	32	160	80
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

Here is what the different columns of the above table stand for:

- The column *Message Size* shows the upper bound on the size of the message that an algorithm can handle.

- The column heading *Block Size* is the size of each bit block that the message is divided into. Recall from Section 15.8 that an input message is divided into a sequence of b-bit blocks. Block size for an algorithm tells us the value of b in the figure on Slide 27.

- The *Word Size* is used during the processing of the input blocks, as will be explained later. *Message Digest Size* refers to the size of the hash code produced.

- Finally, the *Security* column refers to how many messages would have to be generated before one can be found with the same hash code with a probability of 0.5. (This is the Birthday Attack presented in Section 15.7.) As shown previously, in general, for a secure hash algorithm producing n-bit hash codes, one would need to come up with $2^{n/2}$ messages in order to discover a collision with a probability of 0.5. That's why the entries in the last column are half in size compared to the entries in the *Message Digest Size*.

• The algorithms SHA-256, SHA-384, and SHA-512 are collectively referred to as SHA-2.

- Also note that SHA-1 is a successor to MD5 that used to be a widely used hash function.

- SHA-1 was cracked in year 2005 by two different research groups. In particular, Wang, Yin, and Yu demonstrated that it is possible to come up with a collision for SHA-1 with only 2^{69} operations, far fewer than the security level of 2^{80} that is associated with this hash function.

- NIST will withdraw its approval of SHA-1 by 2010.

15.10: The SHA-512 Secure Hash Algorithm

Figure 4 shows the overall processing steps of SHA-512. To describe them in detail:

Append Padding Bits and Length Value: This step makes the input message an exact multiple of 1024 bits:

- The length of the overall message to be hashed must be a multiple of 1024 bits.
- The last 128 bits of what gets hashed are reserved for the message length value.
- This implies that even if the original message were by chance to be an exact multiple of 1024, you'd still need to append another 1024-bit block at the end to make room for the 128-bit message length integer.
- Leaving aside the trailing 128 bit positions, the padding consists of a single 1-bit followed by the required number of 0-bits.

- The length value in the trailing 128 bit positions is an unsigned integer with its most significant byte first.

- The padded message is now an exact multiple of 1024 bit blocks. We represent it by the sequence $\{M_1, M_2, \ldots, M_N\}$, where M_i is the 1024 bits long i^{th} message block.

Initialize Hash Buffer with Initialization Vector: You'll recall from Figure 3 that before we can process the first message block, we need to initialize the hash buffer with IV, the Initialization Vector:

- We represent the hash buffer by **eight 64-bit registers**.

- For explaining the working of the algorithm, these registers are labeled (a, b, c, d, e, f, g, h).

- The registers are initialized by the first 64 bits of the **fractional part of the first eight primes**.

Process Each 1024-bit Message Block M_i: Each message block is taken through 80 rounds of processing. All of this processing is represented by the module labeled f in Figure 4.

- The 80 rounds of processing for each 1024-bit message block are depicted in Figure 5. In this figure, the labels $a, b, c, \ldots, h$ are for the eight 64-bit registers of the **hash buffer**. Figure 5 stands for the modules labeled f in the overall processing diagram in Figure 4.

- In keeping with the overall processing architecture shown in Figure 3, the module f for processing the message block M_i has two inputs: the current contents of the 512-bit hash buffer and the 1024-bit message block. These are fed as inputs to the first of the 80 rounds of processing depicted in Figure 5.

- The round based processing requires a **message schedule** that consists of 80 64-bit words labeled $\{W_0, W_1, \ldots, W_{79}\}$. The first **sixteen** of these, W_0 through W_{15}, are the sixteen 64-bit words in the 1024-bit message block M_i. The rest of the words in the message schedule are obtained by

$$W_i \quad = \quad W_{i-16} \ +_{64} \ \sigma_0(W_{i-15}) \ +_{64} \ W_{i-7} \ +_{64} \ \sigma_1(W_{i-2})$$

where

$$\begin{aligned} \sigma_0(x) &= ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x) \\ \sigma_1(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x) \end{aligned}$$

$$\begin{aligned} ROTR^n(x) &= \textit{circular right shift of the } 64 \textit{ bit arg by } n \textit{ bits} \\ SHR^n(x) &= \textit{left shift of the } 64 \textit{ bit arg by } n \textit{ bits} \\ & \qquad\qquad \textit{with padding by zeros on the right} \\ +_{64} &= \textit{addition module } 2^{64} \end{aligned}$$

- The ith round is fed the 64-bit message schedule word W_i and a special constant K_i.

- The constants K_i's represent the first 64 bits of the **fractional parts of the cube roots of the first eighty prime numbers**. Basically, these constants are meant to be random bit patterns to break up any regularities in the message blocks.

- How the contents of the hash buffer are processed along with the inputs W_i and K_i is referred to as implementing the **round function**.

- The round function consists of a sequence of transpositions and substitutions, all designed to diffuse to the maximum extent possible the content of the input message block. The relationship between the contents of the eight registers of the hash buffer at the input to the ith round and the output from this round is given by

$$
\begin{aligned}
a &= T_1 + T_2 \\
b &= a \\
c &= b \\
d &= c \\
e &= d + T_1
\end{aligned}
$$

$$
\begin{aligned}
f &= e \\
g &= f \\
h &= g
\end{aligned}
$$

where

$$
\begin{aligned}
T_1 &= h \ +_{64} \ Ch(e,f,g) \ +_{64} \ \sum e \ +_{64} \ W_i \ +_{64} \ K_i \\
T_2 &= \sum a \ +_{64} \ Maj(a,b,c) \\
Ch(e,f,g) &= (e \ AND \ f) \oplus (NOT \ e \ AND \ g) \\
Maj(a,b,c) &= (a \ AND \ b) \oplus (a \ AND \ c) \oplus (b \ AND \ c) \\
\sum a &= ROTR^{28}(a) \oplus ROTR^{24}(a) \oplus ROTR^{39}(a) \\
\sum e &= ROTR^{14}(e) \oplus ROTR^{18}(e) \oplus ROTR^{41}(e) \\
+_{64} &= addition \ modulo \ 2^{64}
\end{aligned}
$$

Note that, when considered on a bit-by-bit basis the function $Maj()$ is true, that is equal to the bit 1, only when a majority of its arguments (meaning two out of three) are true. Also, the function $Ch()$ implements at the bit level the conditional statement "if arg1 then arg2 else arg3".

- The output of the 80th round is added to the content of the hash buffer at the beginning of the round-based processing. **This addition is performed separately on each 64-bit word of the output of the 80th modulo 2^{64}.** In other words, the addition is carried out separately for each of the eight registers of the hash buffer modulo 2^{64}.

Finally, ...: After all the N message blocks have been processed (see Figure 4), the content of the hash buffer is the message digest.

15.11: Hash Functions for Computing Message Authentication Codes

- Just as a hash code is a fixed-size fingerprint of a variable-sized message, so is a **message authentication code** (MAC).

- A MAC is also known as a **cryptographic checksum** and as an **authentication tag**.

- A MAC can be produced by appending a secret key to the message and then hashing the composite message. The resulting hash code is the MAC. [A MAC produced with a hash function is also referred to by **HMAC**. A MAC can also be based on a block cipher or a stream cipher. The block-cipher based **DES-CBC MAC** is widely used in various standards.]

- More sophisticated ways of producing a MAC may involve an iterative procedure in which a pattern derived from the key is added to the message, the composite hashed, another pattern derived from the key added to the hash code, the new composite hashed again, and so on.

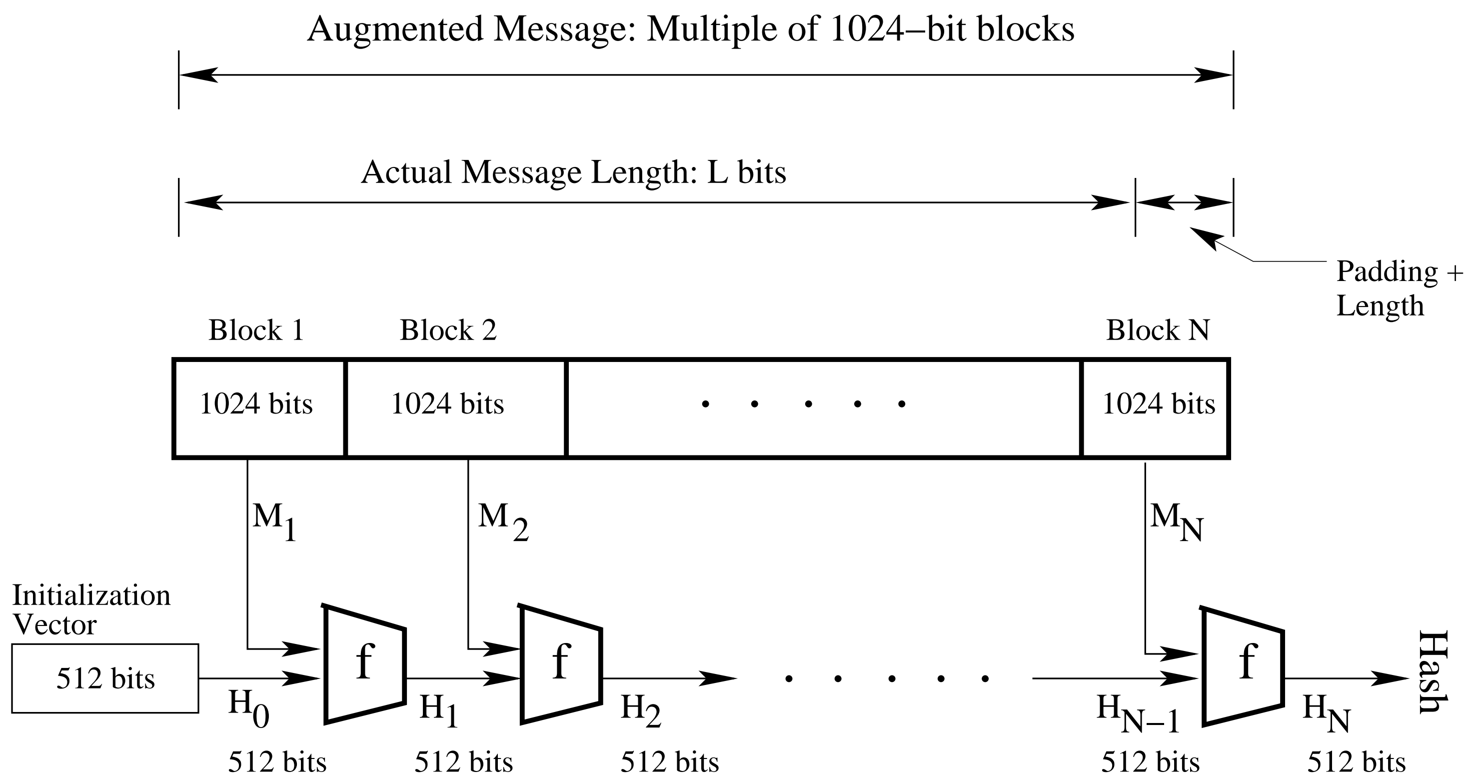


Figure 4: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

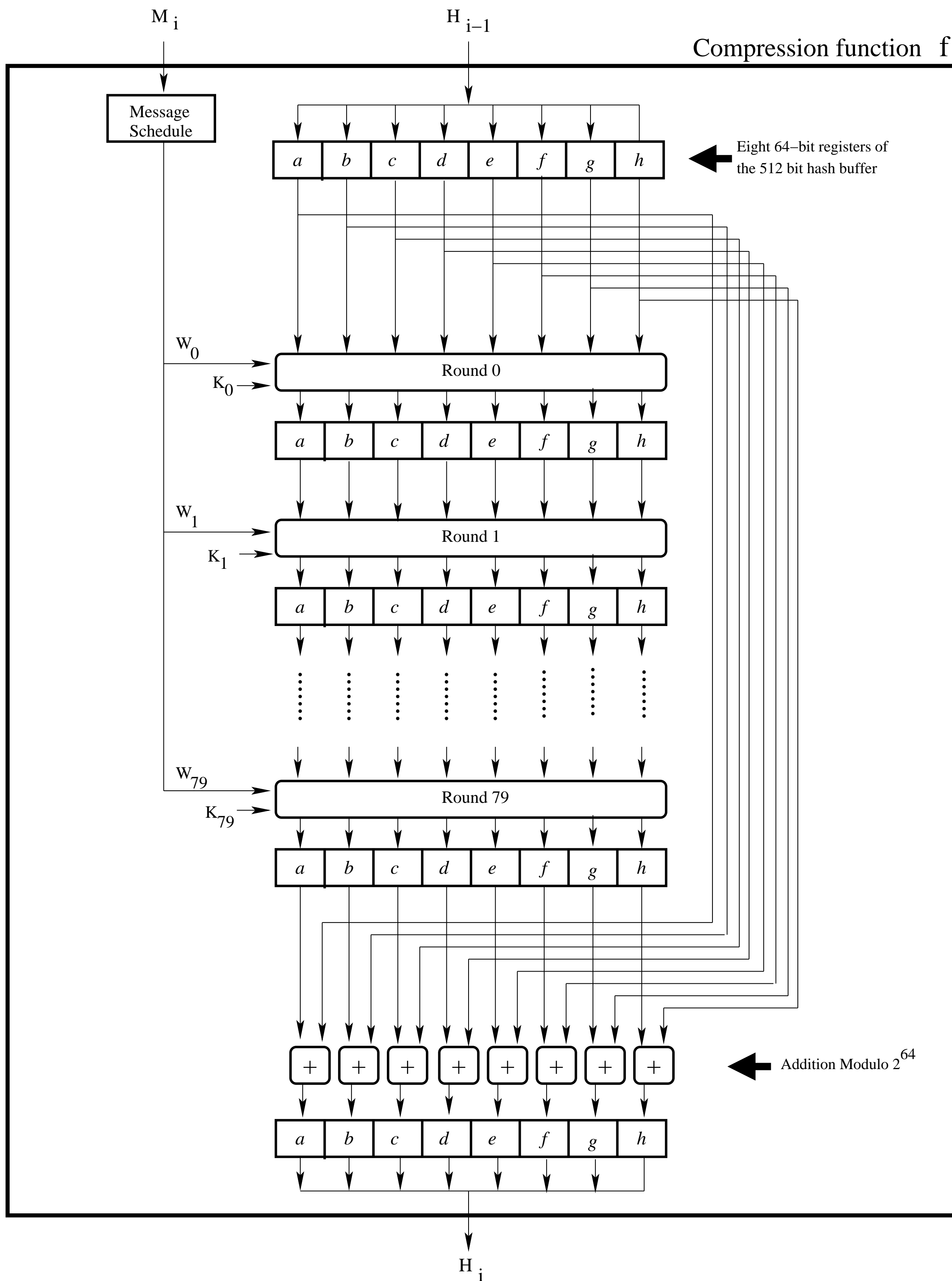


Figure 5: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

- Another way to generate a MAC would be to compress the message into a fixed-size signature and to then encrypt the signature with an algorithm like DES. The output of the encryption algorithm becomes the MAC value and the encryption key the secret that must be shared between the sender and the receiver of a message.

- Assuming a collision-resistant hash function, the original message and its MAC can be safely transmitted over a network without worrying that the integrity of the data may get compromised. A recipient with access to the key used for calculating the MAC can verify the integrity of the message by recomputing its MAC and comparing it with the value received.

- Let's denote the function that generates the MAC of a message M using a secret key K by $C(K, M)$. That is $MAC \quad = \quad C(K,\ M)$.

- Here is a MAC function that is positively **not** safe:

  - Let $\{X_1, X_2, \ldots,\}$ be the 64-bit blocks of a message M. That is $M \quad = \quad (X_1||X_2|| \ldots ||X_m)$. (The operator '$||$' means concatenation.) Let

$$\Delta(M) \quad = \quad X_1 \oplus X_2 \oplus \cdots \oplus X_m$$

– We now define

$$C(K,\ M) \quad = \quad E(K,\ \Delta(M))$$

where the encryption algorithm, $E()$, is assumed to be DES in the electronic codebook mode. (That is why we assumed 64 bits for the block length. We will also assume the key length to be 56 bits.) Let's say that an adversary can observe $\{M, C(K, M)\}$.

– An adversary can easily created a forgery of the message by replacing X_1 through X_{m-1} with **any desired** Y_1 through Y_{m-1} and then replacing X_m with Y_m that is given by

$$Y_m \quad = \quad Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{m-1} \oplus \Delta(M)$$

It is easy to show that when the new message $M_{forged} = \{Y_1||Y_2|| \cdots ||Y_m\}$ is concatenated with the original $C(K, \Delta(M))$, the recipient would not suspect any foul play. When the recipient calculates the MAC of the received message using his/her secret key K, the calculated MAC would agree with the received MAC.

- The lesson to be learned from the unsafe MAC algorithm is that although a brute-force attack to figure out the secret key K would be very expensive (requiring around 2^{56} encryptions of the message), it is nonetheless ridiculously easy to replace a legitimate message with a fraudulent one.

- A commonly-used and cryptographically-secure approach for computing MACs is known as **HMAC**. It is used in the IPSec protocol (for packet-level security in computer networks), in SSL (for transport-level security), and a host of other applications.

- The size of the MAC produced by **HMAC** is the same as the size of the hash code produced by the underlying hash function (which is typically SHA-1).

- The operation of the **HMAC** algorithm is shown Figure 6. This figure assumes that you want an n-bit MAC and that you will be processing the input message M one block at a time, with each block consisting of b bits.

  - The message is segmented into b-bit blocks Y_1, Y_2, ...

  - K is the secret key to be used for producing the MAC.

  - K^+ is the secret key K padded with **zeros on the left** so that the result is b bits long. Recall, b is the length of each message block Y_i.

  - The algorithm constructs two sequences **ipad** and **opad**, the

former by repeating the 00110110 sequence $b/8$ times, and the latter by repeating 01011100 also $b/8$ times.

– The operation of **HMAC** is described by:

$$HMAC_K(M) \quad = \quad h\,(\quad (K \oplus opad) \;||\; h\,(\;(K \oplus ipad) \,||M \;) \quad)$$

where $h()$ is the underlying iterated hash function of the sort we have covered in this lecture.

- The security of **HMAC** depends on the security of the underlying hash function, and, of course, on the size and the quality of the key.

- For further information on **HMAC**, see Chapter 12 of "Cryptography and Network Security" by William Stallings, the source of the information presented here.

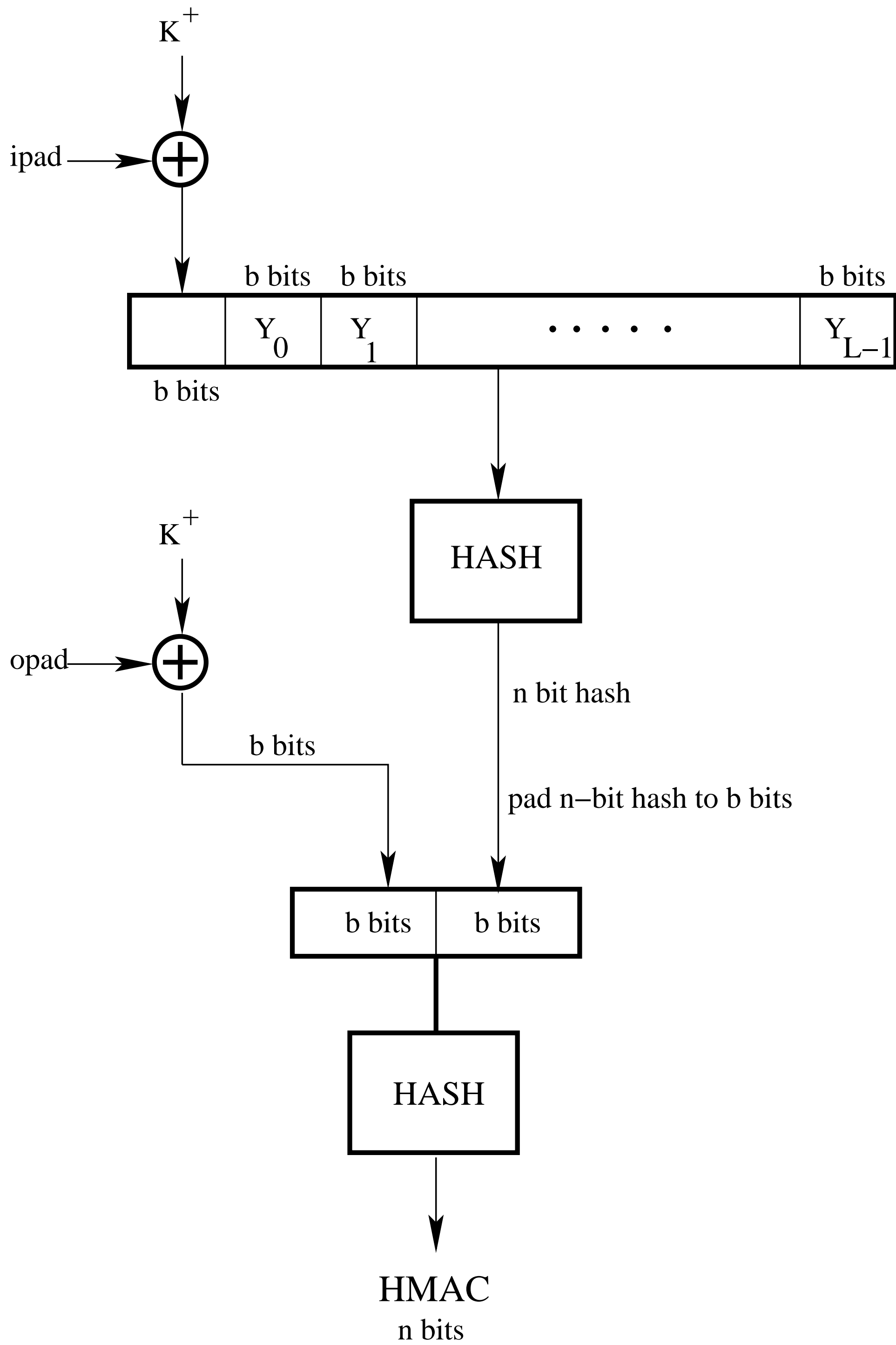


Figure 6: *This figure is from "Computer and Network Security" by Avi Kak*

Acknowledgement

Prateek Singhal caught a couple of typographical errors in the equations on slide 26. Thanks Prateek.