Lecture 15: Hashing for Message Authentication

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 26, 2013
5:46pm

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
- **A compact Python implementation for SHA-1 using BitVector**
- Message Authentication Codes

CONTENTS

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**. [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 (or the message digest) 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 an innocuous (practically invisible) change in a message, here is an example:

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

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

  The only difference between the two messages shown above is the extra space between the words "hungry" and "brown" in the second message. Notice how completely different the hash codes look. SHA-1 produces a 160 bit hash code. It takes 40 hex characters to show the code in hex.

- The two hash codes (or, message digests, if you would rather call them that) shown above 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, used in the script shown above, can be used to invoke any of over fifteen different 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. A similar functionality in Python is provided by the **hashlib** library. Both the **Digest** module for Perl and the **hashlib** library for Python come with the standard distribution of the two 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, as explained previously in Lecture 13.**

- 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.

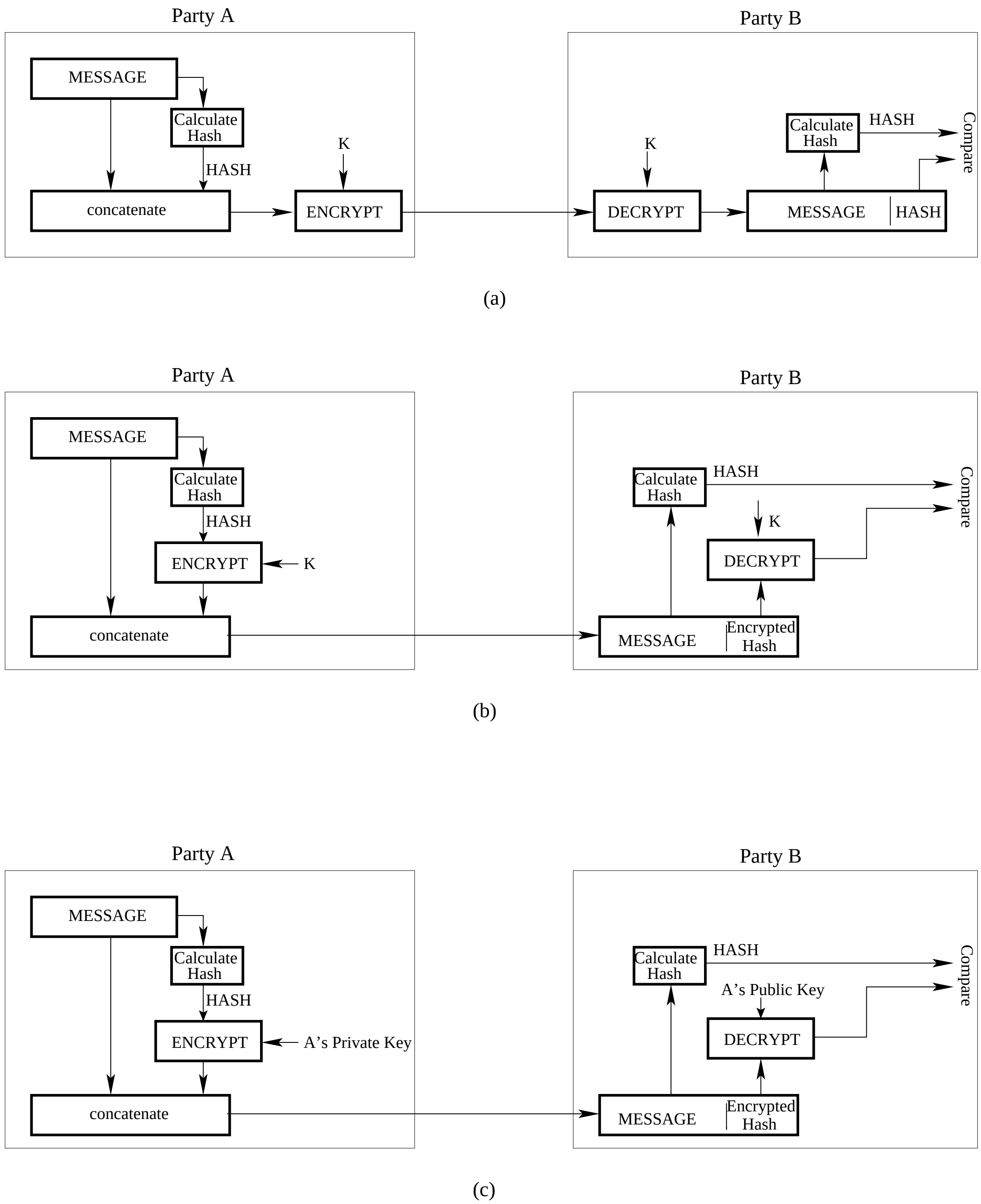

Figure 1: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

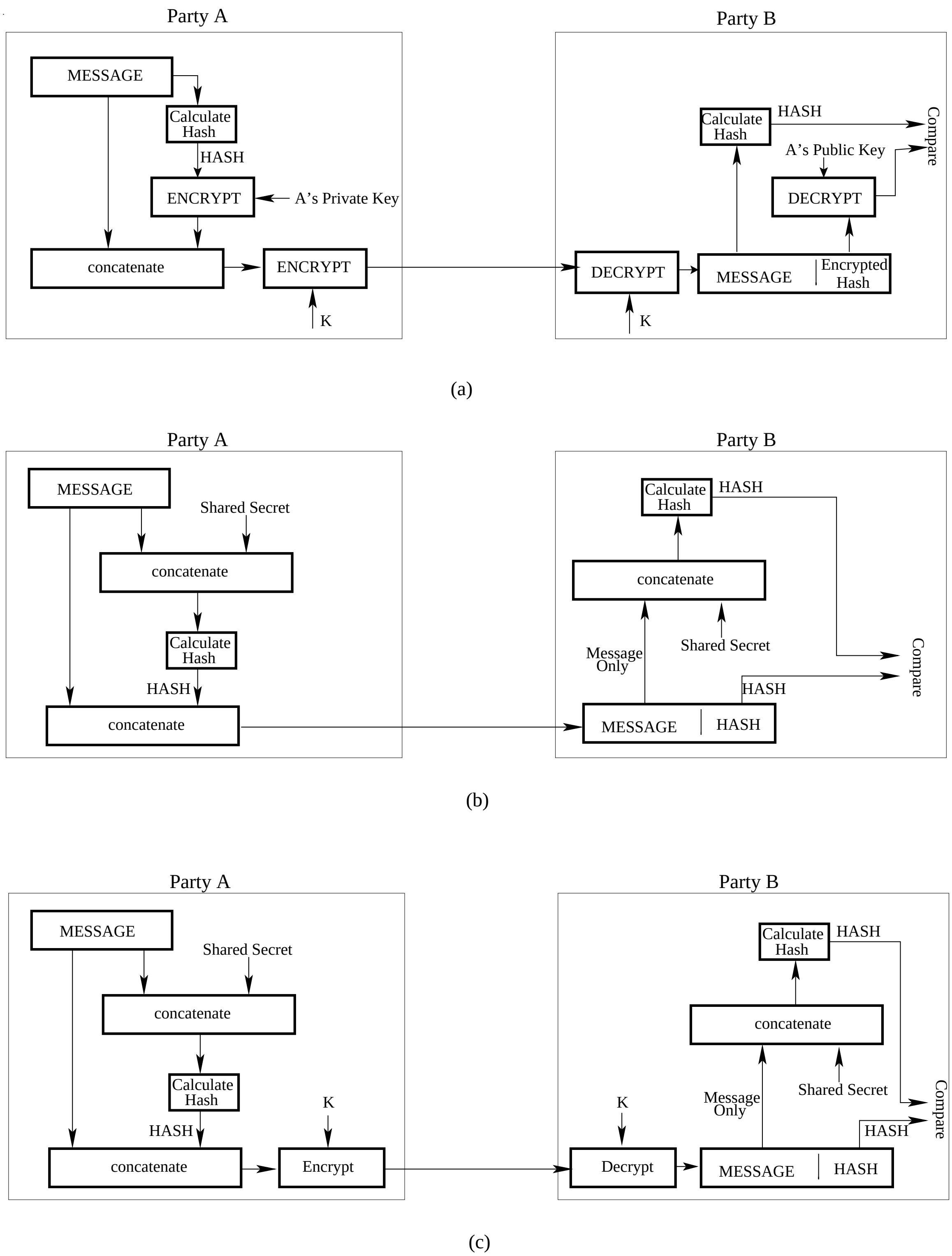

Figure 2: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

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 no constraints whatsoever on the messages and if there can be an arbitrary number of different possible 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.

- There exist several applications, such as in the dissemination of popular media content, where confidentiality of the message content is not an issue, but authentication is. In such applications, we would like to to send unencrypted plaintext messages with encrypted hash codes. This would eliminate the computational overhead of encryption and decryption for the main message content and yet allow for authentication. But this would work only if the hashing function has perfect collision resistance. 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.

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 VALUE?

- Assume that we have a random message generator and that we can calculate the hash code for each message produced by the generator.

- Let's say we are interested in whether any of the messages is going to have its hash code equal to a particular value h.

- Let's consider a pool of k messages produced randomly by the message generator.

- We pose the following question: What is the value of k so that the pool contains **at least one** message whose hash code is equal to h with probability 0.5?

- To find k, we reason as follows:

  – Let's say that the hash code can take on N different but equiprobable values.

  – Say we pick a message x at random from the pool of messages. Since all N hash codes are equiprobable, the probability of message x having its hash code equal to h is $\frac{1}{N}$.

  – Since the hash code of message x either equals h or does not equal h, the probability of the latter is $1 - \frac{1}{N}$.

  – If we pick, say, two messages x and y randomly from the pool, the events that the hash code of neither is equal to h are probabilistically independent. That implies that the probability that **none** of two messages has its hash code equal to h is $(1 - \frac{1}{N})^2$. [Of course, by similar reasoning, the probability that **both** x and y will have their hash codes equal to h is $(\frac{1}{N})^2$. But it is more difficult to use such joint probabilities to answer our overall question stated in red on the previous page on account of the phrase "at least one" in it. Also see the note in blue at the end of this section.]

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

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

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

– 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

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

- 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 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 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 with a probability of 0.5.

- To illustrate the danger of arriving at formulas through back-of-the-envelope reasoning, consider the following seemingly more straightforward approach to the derivation of Equation (2): With all hash codes being equiprobable, the probability that any given message has its hash code equal to a particular value h is obviously $1/N$. Now consider a pool of just 2 messages. Speaking colloquially (that is, without worrying about violating the rules of logic), as you might over a glass of wine in a late-night soiree, the event that this pool has at least one message whose hash code is h is made up of the event that the first of the two messages has its hash code equal to h **or** the event that the second of the two messages has its hash code equal to h. Since the two events are disjunctive, the probability that a pool of two messages has at least one message whose hash code is h is a sum of the individual probabilities in the disjunction — that gives is a probability of $2/N$. Generalizing this argument to a pool of k messages, we get for the desired probability a value of k/N that was shown in Equation (2). But this formula, if considered as a precise formula for the probability we are looking for, couldn't possibly be correct. As you can see, this formula gives us absurd values for the probability when k is equal to or exceeds N.

15.5.1: What Is the Probability That There Exist At Least Two Messages With the Same Hash Code?

- Given a pool of k messages, the question "*What is the probability that there exists at least one message in the pool whose hash code is equal to a specific value?*" **is very different from the question** "*What is the probability that there exist at least two messages in the pool whose hash codes are the same?"*

- Raising the same two questions in a different context, the question "*What is the probability that, in a class of* 20 *students, someone else has the same birthday as yours (assuming you are one of the* 20 *students)?*" **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 former question was addressed in the previous section. Based on the result derived there, the probability of the former is approximately $\frac{19}{365}$. The latter question we will address in this section. As you will see, the probability of the latter is roughly the much larger value $\frac{(20\times 19)/2}{365} = \frac{190}{365}$. This is referred to as the **birthday paradox**, paradox only in the sense that it seems counterintuitive. [A quick way to accept the 'paradox' intuitively is that you can construct $C(20,2) = \binom{20}{2} = \frac{20!}{18!2!} = \frac{20\times 19}{2} = 190$ different possible pairs from a group of 20 people. Since this number, 190, is rather comparable to 365, the total number of different birthdays, the conclusion is not surprising.] The birthday paradox states that given a group of 23 or more randomly chosen people, the probability that at least two of them will have the same birthday is more than 50%. And if we randomly choose 60 or more people, this probability is greater than 90%. [A man on the street would certainly think that it would take many more than 60 people for any two of them to have the same birthday with near certainty. That's why we refer to this as a 'paradox.' Note, however, it is NOT a paradox in the sense of being a logical contradiction.]

- 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 of messages with the same hash code 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 first find out 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 first message in the pool, we can choose any arbitrarily. Since there exist only N distinct hash codes, and, therefore, since there can only be N different messages with distinct hash codes, 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, for the second entry in the pool, we can select a message corresponding to the other $N-1$ still available hash codes.

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

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

– 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

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

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.

- When k is small and N large, we can use the approximation $e^{-x} \approx 1 - x$ in the above formula and express it as

$$1 \;-\; \left(1 \;-\; \frac{k(k-1)}{2N}\right) \;\;=\;\; \frac{k(k-1)}{2N} \tag{13}$$

It was this formula that we used when we mentioned the birthday paradox at the beginning of this section. There we had $k = 20$ and $N = 365$.

- We will now use Equation (12) 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

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

Simplifying, we get

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

Therefore,

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

which gives us

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

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

$$k^2 \quad \approx \quad (2ln2)N \tag{14}$$

implying

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

- 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 pool of $2^{n/2}$ randomly generated messages will contain at least one pair of messages with the same 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.

15.6: THE BIRTHDAY ATTACK

- This attack applies to the following scenario: Say Mr. BigShot has a dishonest assistant, Mr. Creepy, preparing contracts for Mr. BigShot's digital signature.

- Mr. Creepy prepares the legal contract for a transaction. Mr. Creepy 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.

- Next, Mr. Creepy prepares a fraudulent version of the contract. As with the correct version, Mr. Creepy 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 of, as you'd expect, Mr. Creepy getting Mr. BigShot to digitally sign a correct version of the contract, meaning getting Mr. BigShot to encrypt the hash code of the correct version of the contract with his private key, and then replacing the contract by its fraudulent version that has the same hash code value.

- This attack is called the birthday attack because the combinatorial issues involved are the same as in the birthday paradox presented earlier in Section 15.5.1. Also note that for n-bit hash coding, the approximate value we obtained for k is the same in both cases. That is, $k = 2^{n/2}$.

15.7: 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. Additionally, 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 — even for very short messages. [See Section 15.3 for the two important properties of secure hash functions. We are talking about the same two properties here. "Secure" and "cryptographically secure" mean the same thing for hash functions.]

- Most secure hash functions are based on the structure proposed by Ralph Merkle in 1979. This structure forms the basis of MD5, Whirlpool and the SHA series of hash functions.

- The input message is partitioned into L number of 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.

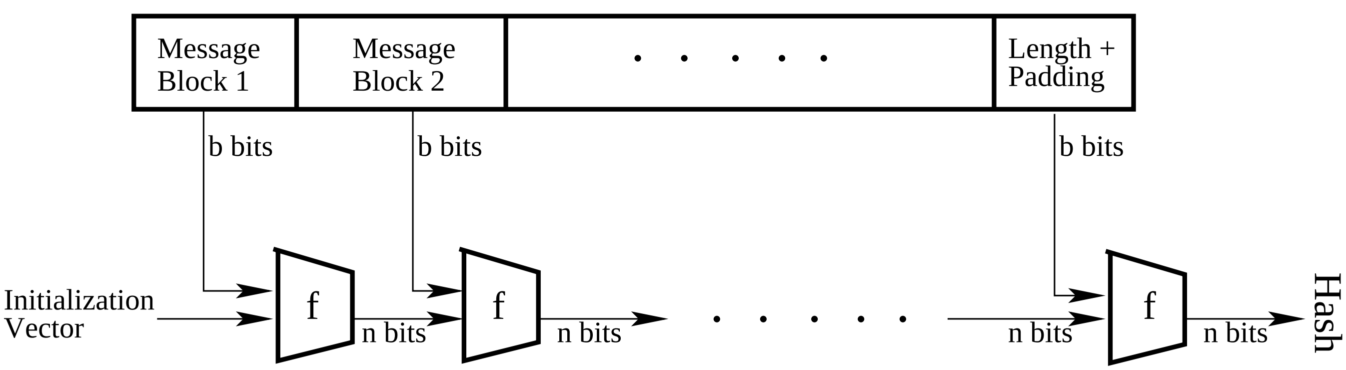

Figure 3: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

15.7.1: 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 15.7 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 Figure 3.

– The *Word Size* is used during the processing of the input blocks, as will be explained later.

– The *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 based on the Birthday Attack arguments presented in Section 15.6.] 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 was a widely used hash function. There still exist many legacy applications that use MD5 for calculating hash codes.

- SHA-1 was cracked in the year 2005 by two different research groups. In one of these two demonstrations, Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu demonstrated that it was possible to come up with a collision for SHA-1 within a space of size only 2^{69}, which was far fewer than the security level of 2^{80} that is associated with this hash function.

- I believe that, in 2010, NIST officially withdrew its approval of SHA-1 for applications that need to be compliant with U.S. Government standards.

- All of the SHA family of hash functions are described in the `FIPS180` document that can be downloaded from:

  `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`

  The SHA-512 algorithm details presented in the next subsection are taken from the above document.

15.7.2: 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.

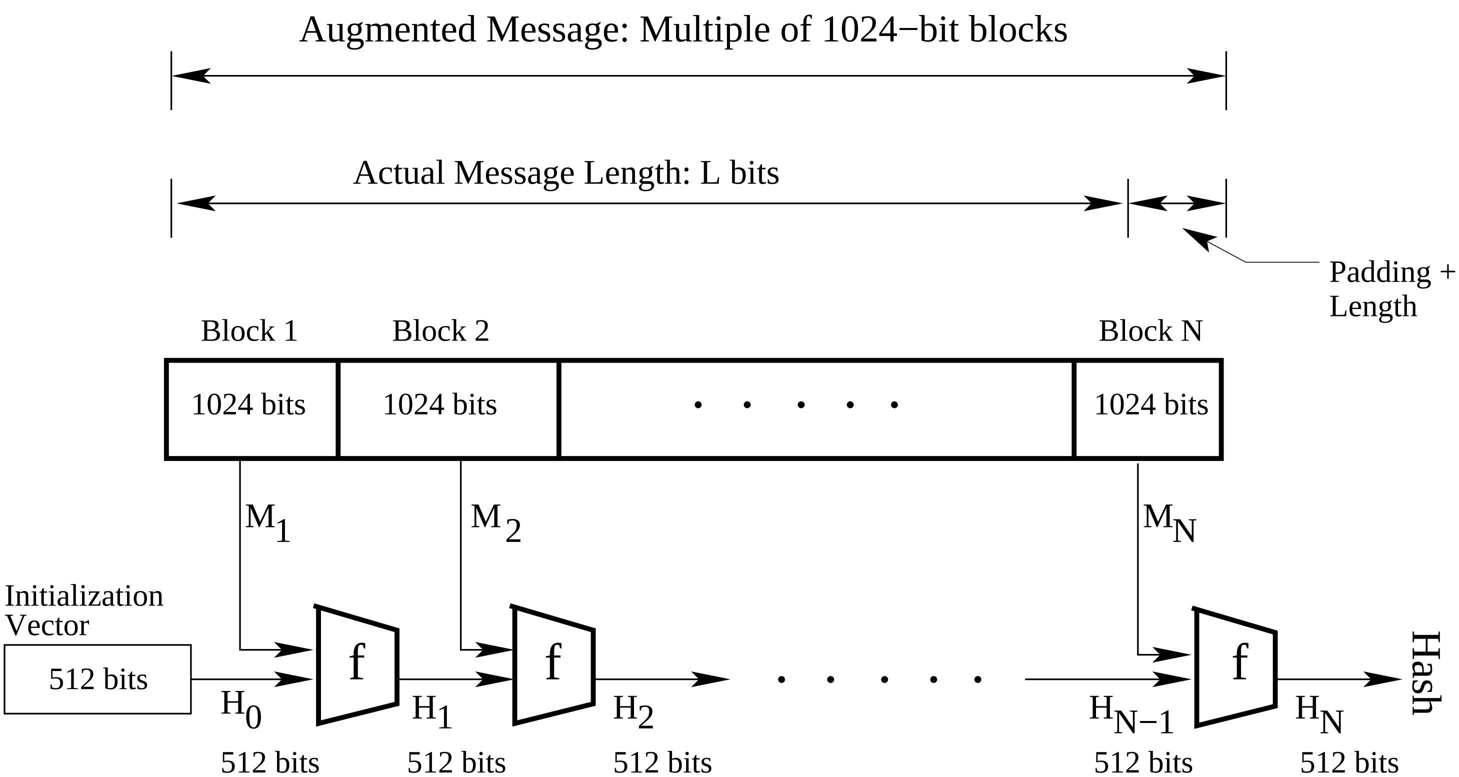

Figure 4: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

- 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 parts of the square-roots of the first eight primes**. These are shown below in hex:

```
6a09e667f3bcc908
bb67ae8584caa73b
3c6ef372fe94f82b
a54ff53a5f1d36f1
510e527fade682d1
```

```
9b05688c2b3e6c1f
1f83d9abfb41bd6b
5be0cd19137e2179
```

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

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

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

$$+_{64} \quad = \quad addition\ module\ 2^{64}$$

- The i^{th} 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. These constants are shown below in hex. They are to be read from left to right and top to bottom. [In other words, K_0 is the first value in the first row, K_1 the second value in the first tow, K_2 the third value in the first row, K_3 the last value in the first row. For K_4, we look at the first value in the second row; and so on.]

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

- 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 i^{th} round and the output from this round is given by

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

$$
\begin{aligned}
f &= e \\
e &= d \ +_{64} \ T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 \ +_{64} \ T_2
\end{aligned}
$$

where $+_{64}$ again means modulo 2^{64} addition and where

$$
\begin{aligned}
T_1 &= h \ +_{64} \ Ch(e,f,g) \ +_{64} \ \sum e \ +_{64} \ W_i \ +_{64} \ K_i \\
T_2 &= \sum a \ +_{64} \ Maj(a,b,c) \\
Ch(e,f,g) &= (e \ AND \ f) \oplus (NOT \ e \ AND \ g) \\
Maj(a,b,c) &= (a \ AND \ b) \oplus (a \ AND \ c) \oplus (b \ AND \ c) \\
\sum a &= ROTR^{28}(a) \oplus ROTR^{34}(a) \oplus ROTR^{39}(a) \\
\sum e &= ROTR^{14}(e) \oplus ROTR^{18}(e) \oplus ROTR^{41}(e) \\
+_{64} &= addition \ modulo \ 2^{64}
\end{aligned}
$$

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 80^{th} 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** 80^{th} **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.

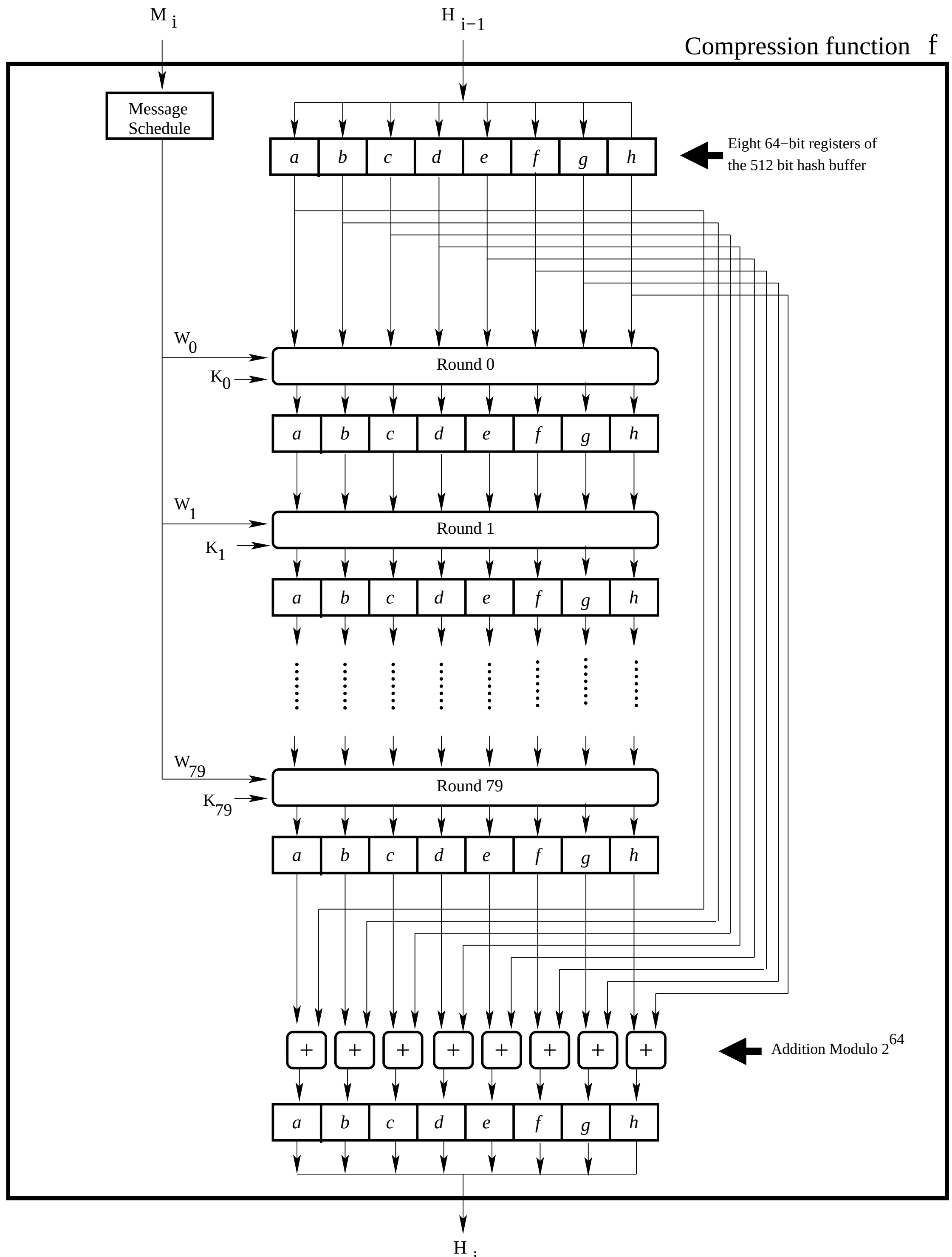

Figure 5: *This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*

15.7.3: A Compact Python Implementation for SHA-1 Using BitVector

- My goal in this section is to demonstrate a Python implementation for SHA-1 in order to help you do the same for SHA-512 in the second of the programming homeworks at the end of this lecture.

- Even more specifically, my goal here is to show you how the BitVector module in Python can be used for creating a compact program for a cryptographically secure hashing algorithms. A typical implementation of a Python script that calculates one of the SHA hashcodes is typically around 300 lines of code. With BitVector, you can do the same in about 100 lines of code.

- Since you already know about SHA-512, let me first quickly present the highlights of SHA-1 so that you can make sense of the Python code that follows.

- Whereas SHA-512 used a block length of 1024 bits, SHA-1 uses a block length of 512 bits. After padding and incorporation of the length of the original message, what actually gets hashed must be integral multiple of 512 bits in length. Just as in SHA-512, we first extend the message by a single bit '1' and then insert an appropriate number of 0 bits until we are left with just 64 bit positions at the end in which we place the length of the original message in big endian representation. Since the length field is 64 bits long, obviously, the longest message that is meant to be hashed by SHA-1 is 2^{64} bits.

- Let's say that L is the length of the original message. After we extend the message by a single bit '1', the length of the extended message is $L + 1$. Let N be the number of zeros needed to append to the extended message so that we are left with 64 bits at the end where we can store the length of the original message. The following relationship must hold: $(L + 1 + N + 64)\ \%\ 512 = 0$ where the Python operator % carries out division of the left operand modulo the right operand and returns the remainder. This implies that $N = 448 - (L + 1)\ \%\ 512$. [The reason for sticking 1 at the end of a message is to be able to deal with empty messages. So when the original message is an empty string, the extended message will still consist of a single 1 bit.]

- As in SHA-512, each block of 512 bits is taken through 80 rounds of processing. A block is divided into 16 32-bit words for round-based processing. In the code shown at the end of this section, we denote these 16 words by $w[i]$ for i from 0 through 15. These 16 words extracted from a block are extended into an 80 word schedule by the formula:

$$w[i] \;=\; w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]$$

for i from 16 through 79.

- The initialization vector needed for the first invocation of the compression function is given by a concatenation of the following five 32-bit words:

$$\begin{aligned} h0 &= 67452301 \\ h1 &= efcdab89 \\ h2 &= 98badcfe \\ h3 &= 10325476 \\ h4 &= c3d2e1f0 \end{aligned}$$

  where each of the five parts is shown as a sequence of eight hex digits.

- The goal of the compression function for each block of 512 bits of the message is to process the 512 block along with the 160-bit hash code produced for the previous block to output the 160-bit hashcode for the new block. The final 160-bit hashcode is the SHA-1 digest of the message.

- As mentioned, the compression function for each 512-bit block works in 80 rounds. These rounds are organized into 4 round sequences of 20 rounds each, with each round sequence characterized by its own processing function and its own round constant. If the five 32-words on the hashcode produced by the previous 512-bit block are denoted a, b, c, d, and e, then for the first 20 rounds the function and the round constant are given by

$$\begin{aligned} f &= (b \;\&\; c) \oplus \Big((\sim b) \;\&\; d\Big) \\ k &= 0x5a827999 \end{aligned}$$

  For the second 20 round-sequence the function and the constant are given by

$$\begin{aligned} f &= b \;\oplus\; c \;\oplus\; d \\ k &= 0x6ed9eba1 \end{aligned}$$

  The same for the third 20 round-sequence are given by

$$\begin{aligned} f &= (b \;\&\; c) \;\oplus\; (b \;\&\; d) \;\oplus\; (c \;\&\; d) \\ k &= 0x8f1bbcdc \end{aligned}$$

  And, for the fourth and the final 20 round sequence, we have

$$\begin{aligned} f &= b \;\oplus\; c \;\oplus\; d \\ k &= 0xca62c1d6 \end{aligned}$$

- At the i^{th} round, $i = 0 \ldots 79$, we update the values of a, b, c, d, and e by first calculating

$$T \;=\; \Big((a << 5) \;+\; f + \;e \;+\; k \;+\; w[i]\Big) \; mod \; 2^{32}$$

where $w[i]$ is the i^{th} word in the 80-word schedule obtained from the sixteen 32-words of the message block. Next, we update the values of a, b, c, d, and e as follows

$$\begin{aligned} e &= d \\ d &= c \\ c &= b << 30 \\ b &= a \\ a &= T \end{aligned}$$

where you have to bear in mind that while c is set to b circularly rotated to the left by 30 positions, but the value of b itself must remain unchanged for the logic of SHA1. This is particularly important in light of how b is used at the end of 80 rounds of processing for a 512-bit message block.

- After all of the 80 rounds of processing are over, we create output hash code for the current 512-bit block of the message by

$$\begin{aligned} h0 &= (h0 + a) \ mod \ 2^{32} \\ h1 &= (h1 + b) \ mod \ 2^{32} \\ h2 &= (h2 + c) \ mod \ 2^{32} \\ h3 &= (h3 + d) \ mod \ 2^{32} \\ h4 &= (h4 + e) \ mod \ 2^{32} \end{aligned}$$

Note that each hi is a 32 bit word. The hashcode produced after the current block has been processed is the concatenation of $h0$, $h1$, $h2$, $h3$, and $h4$. This hashcode produced after the final message block is processed is the SHA1 hash of the input message.

- The implementation shown below is meant to be invoked in a command-line mode as follows:

```
sha1_from_command_line.py   string_whose_hash_you_want
```

- The implementation follows.

```
#!/usr/bin/env python

##  sha1_from_command_line.py
##  by Avi Kak (kak@purdue.edu)
##  Feb 19, 2013

##  Call syntax:
##
##     sha1_from_command_line.py  string_to_be_hashed

##  This script takes its message on the standard input from
##  the command line and sends the hash to its standard
##  output.  NOTE: IT ADDS A NESWLINE AT THE END OF THE OUTPUT
##  TO SHOW THE HASHCODE IN A LINE BY ITSELF.

import sys
import BitVector
if BitVector.__version__ < '3.2':
    sys.exit("You need BitVector module of version 3.2 or higher" )
from BitVector import *

if len(sys.argv) != 2:
    sys.stderr.write("Usage: %s  <string to be hashed>\n" % sys.argv[0])
    sys.exit(1)
```

```
message = sys.argv[1]

# Initialize hashcode for the first block. Subsequetnly, the
# output for each 512-bit block of the input message becomes
# the hashcode for the next block of the message.
h0 = BitVector(hexstring='67452301')
h1 = BitVector(hexstring='efcdab89')
h2 = BitVector(hexstring='98badcfe')
h3 = BitVector(hexstring='10325476')
h4 = BitVector(hexstring='c3d2e1f0')

bv = BitVector(textstring = message)
length = bv.length()
bv1 = bv + BitVector(bitstring="1")
length1 = bv1.length()
howmanyzeros = (448 - length1) % 512
zerolist = [0] * howmanyzeros
bv2 = bv1 + BitVector(bitlist = zerolist)
bv3 = BitVector(intVal = length, size = 64)
bv4 = bv2 + bv3
words = [None] * 80
for n in range(0,bv4.length(),512):
    block = bv4[n:n+512]
    words[0:16] = [block[i:i+32] for i in range(0,512,32)]
    for i in range(16, 80):
        words[i] = words[i-3] ^ words[i-8] ^ words[i-14] ^ words[i-16]
        words[i] << 1
        a,b,c,d,e = h0,h1,h2,h3,h4
    for i in range(80):
        if (0 <= i <= 19):
            f = (b & c) ^ ((~b) & d)
            k = 0x5a827999
        elif (20 <= i <= 39):
            f = b ^ c ^ d
            k = 0x6ed9eba1
        elif (40 <= i <= 59):
            f = (b & c) ^ (b & d) ^ (c & d)
            k = 0x8f1bbcdc
        elif (60 <= i <= 79):
            f = b ^ c ^ d
            k = 0xca62c1d6
        a_copy = a.deep_copy()
        T = BitVector( intVal = (int(a_copy << 5) + int(f) + \
                   int(e) + int(k) + int(words[i])) % (2 ** 32), size=32 )
```

```
        e = d
        d = c
        b_copy = b.deep_copy()
        b_copy << 30
        c = b_copy
        b = a
        a = T
    h0 = BitVector( intVal = (int(h0) + int(a)) % (2**32), size=32 )
    h1 = BitVector( intVal = (int(h1) + int(b)) % (2**32), size=32 )
    h2 = BitVector( intVal = (int(h2) + int(c)) % (2**32), size=32 )
    h3 = BitVector( intVal = (int(h3) + int(d)) % (2**32), size=32 )
    h4 = BitVector( intVal = (int(h4) + int(e)) % (2**32), size=32 )
message_hash = h0 + h1 + h2 + h3 + h4
hash_hex_string = message_hash.getHexStringFromBitVector()
sys.stdout.writelines((hash_hex_string, "\n"))
```

15.8: 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.

- 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 $M = (X_1||X_2||\ldots||X_m)$. (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.

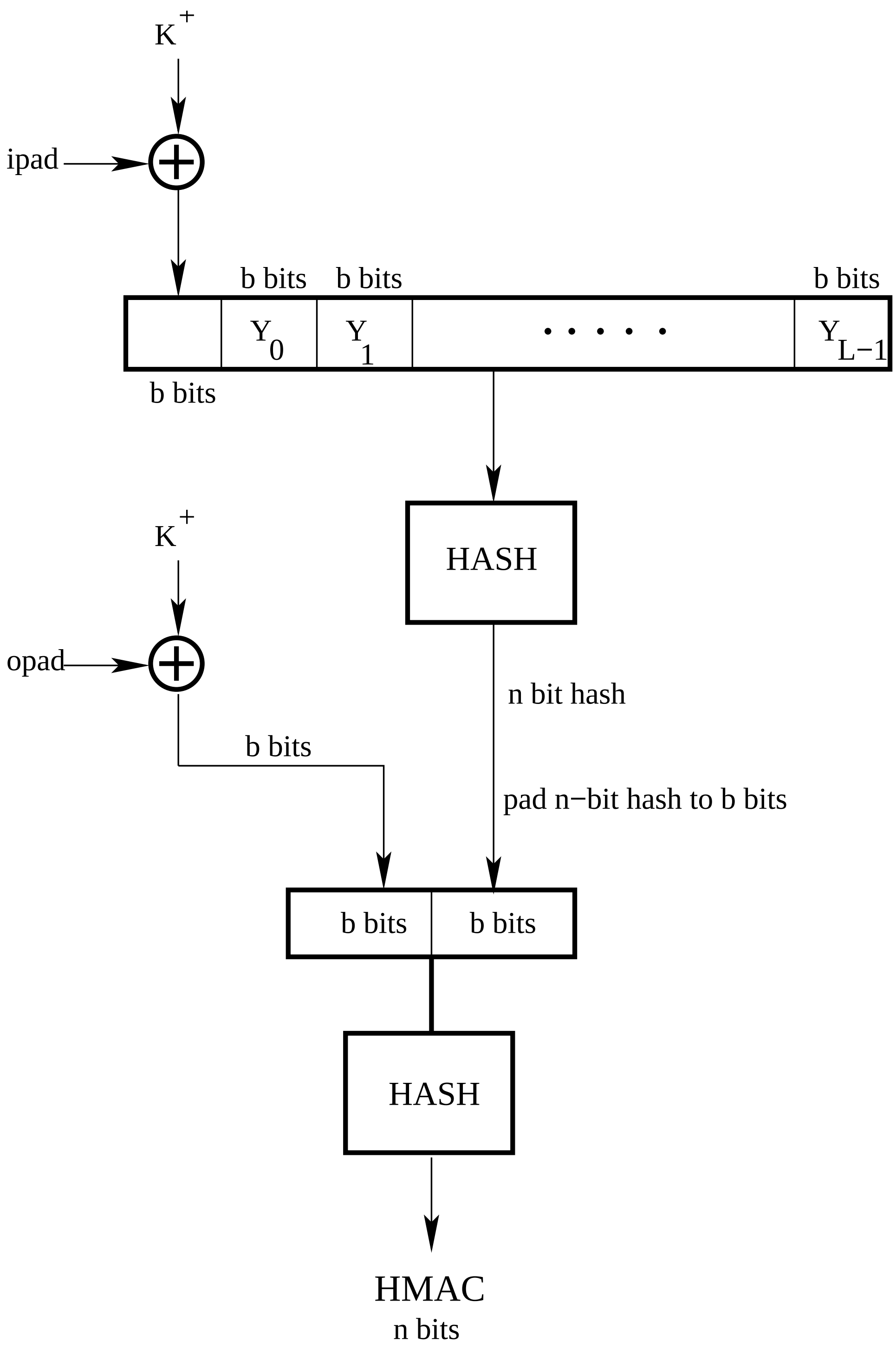

Figure 6: *This figure is from "Computer and Network Security" by Avi Kak*

15.9: HOMEWORK PROBLEMS

1. What is a hash code?

2. If you had only one minute to write a program that calculates the 8-bit hash code of the contents of a disk file, how might you do it?

3. Why would is it a foolish exercise to calculate an 8-bit hash by XORing all the bytes in a file?

4. Even though its support will soon be withdrawn by the government, what is probably the most frequently used hash coding algorithm used today? What is the size of the hash code produced by this algorithm?

5. 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 non-negative solution to

$$L + 1 + K \equiv 448 \pmod{512}$$

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

```
     Message      =     "abc"
     length L     =     24 bits

 01100001  01100010   01100011   1 00......000   00...011000

    a         b        c            <---423--->   <---64---->

 <-------------------   512  ----------------------------->
```

Now here is the question: Why do we include the length of the message in the calculation of the hash code?

6. The fact that only the last 64 bits of the padded message are used for representing the length of the message implies that SHA1 should NOT be used for messages that are longer than what?

7. SHA1 scans through a document by processing 512-bit blocks. Each block is hashed into a 160 bit hash code that is then used as the initialization vector for the next block of 512 bits. This obviously requires a 160 bit initialization vector for the first 512-bit block. Here is the vector:

```
         H_0    =    67452301         (32 bits in hex)
```

```
H_1    =    efcdab89
H_2    =    98badcfe
H_3    =    10325476
H_4    =    c3d2e1f0
```

How are these numbers selected?

8. Why can a hash function not be used for encryption?

9. What is meant by the strong collision resistance property of a hash function?

10. Right or wrong: When you create a new password, only the hash code for the password is stored. The text you entered for the password is immediately discarded.

11. What is the relationship between "hash" as in "hash code" or "hashing function" and "hash" as in a "hash table"?

12. **Programming Assignment:**

    To gain further insights into hashing, the goal of this homework is to implement in Perl or Python a very simple hash function (that is meant more for play than for any serious production work). Write a function that creates a 32-bit hash of a file through the following steps: (1) Initialize the hash to all zeros; (2) Scan the file one byte at a time; (3) Before a new byte is read from the file, circularly shift the bit pattern in the hash to the left by four positions; (4) Now XOR the new byte read from the file with the least significant byte of the hash. Now scan your directory (a very simple thing to do in both Perl and Python, as shown in Chapters 2 and 3 of my SwO book) and compute the hash of all your files. Dump the hash values in some output file. Now write another two-line script to check if your hashing function is exhibiting any collisions. Even though we have a trivial hash function, it is very likely that you will not see any collisions even if your directory is large. Subsequently, by using a couple of files (containing random text) created specially for this demonstration, show how you can make their hash codes to come out to be the same if you alter one of the files by appending to it a stream of bytes that would be the XOR of the original hash values for the files (after you have circularly rotated the hash value for the first file by 4 bits to the left). *NOTE: This homework is easy to implement in Python if you use the BitVector class.*

13. **Programming Assignment:**

    In a manner similar to what I demonstrated in Section 15.7.3 for SHA-1, this homework calls on you to implement the SHA-512 algorithm using the facilities provided by the BitVector module.