Lecture 15: Hashing for Message Authentication

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 24, 2019
12:35am

Goals:

- The birthday paradox and the birthday attack
- Structure of cryptographically secure hash functions
- SHA series of hash functions
- **Compact Python and Perl implementations for SHA-1 using BitVector** [Although SHA-1 is now considered to be fully broken (see Section 15.7.1), programming it is still a good exercise if you are learning how to code Merkle type hash functions.]
- **Compact Python implementation of SHA-256 using BitVector**
- **Crypto Currencies and Their Use of Hash Functions**
- Hash functions for the **other** major application of hashing — Efficient storage of associative data

CONTENTS

15.1: WHAT IS A HASH FUNCTION?

- In the context of message authentication, a hash function takes a **variable sized input message** and produces a **fixed-sized output**. The output is usually referred to as the **hashcode** or the **hash value** or the **message digest**. [Hash functions are also extremely important for creating efficient storage structures for associative arrays in the memory of a computer. (As to what is meant by an "associative array", think of a telephone directory that consists of *<name,number>* pairs.) Those types of hash functions also play a central role in many modern big-data processing algorithms. For example, in the MapReduce framework used in Hadoop, a hash function is applied to the "keys' related to the Map tasks in order to determine their bucket addresses, with each bucket constituting a Reduce task. In this lecture, the notion of a hash function for efficient storage is briefly reviewed in Section 15.10.]

- 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 hashcode (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 hashcode produced by an innocuous (practically invisible) change in a message, here is an example:

```
Message:           "The quick brown fox jumps over the lazy dog"
SHA1 hashcode:     2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

Message:           "The quick brown  fox jumps over the lazy dog"
SHA1 hashcode:     8de49570b9d941fb26045fa1f5595005eb5f3cf2
```

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

- The two hashcodes (or, message digests, if you would rather call them that) shown above were produced by the following interactive session with Python:

```
>>>   import hashlib
>>>   hasher = hashlib.sha1()
>>>   hasher.update("The quick brown fox jumps over the lazy dog")
>>>   hasher.hexdigest()
'2fd4e1c67a2d28fced849ee1bb76e7391b93eb12'
>>>   hasher = hashlib.sha1()
>>>   hasher.update("The quick brown  fox jumps over the lazy dog")
>>>   hasher.hexdigest()
'8de49570b9d941fb26045fa1f5595005eb5f3cf2'
```

As the session shows, I used Python's very popular **hashlib** module for its **sha1** hashing function. Note that I had to call **hashlib.sha1()** again for the second try at hashing. That is because repeated calls to **hasher.update()** concatenate the string arguments supplied to the **update()** function.

- What I have show above with interactive Python can be done in Perl with the following simple script:

```
#!/usr/bin/perl -w
use Digest::SHA1;
my $hasher =  Digest::SHA1->new();
$hasher->add( "The quick brown fox jumps over the lazy dog" );
print $hasher->hexdigest;
print "\n";
$hasher->add( "The quick brown  fox jumps over the lazy dog" );
print $hasher->hexdigest;
print "\n";
```

- Both the Python's **hashlib** module and the Perl's **Digest** module used in the above script can be used to invoke any of over fifteen different hash algorithms. The modules can output the hashcode in either binary format, or in hex format, or a binary string output as in the form of a Base64-encoded string.

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 hashcode 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 hashcode, which is then compared with the hashcode calculated from the received message. The hashcode 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 hashcode 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 hashcode for the message. So the receiver can verify whether or not the message is authentic. [A hashcode produced in the manner shown in Figure 1(b) is also known as the **Message Authentication Code (MAC)** and the overall hash function as a **keyed hash function**. We will discuss such applications of hash functions in greater detail in Section 15.8.]

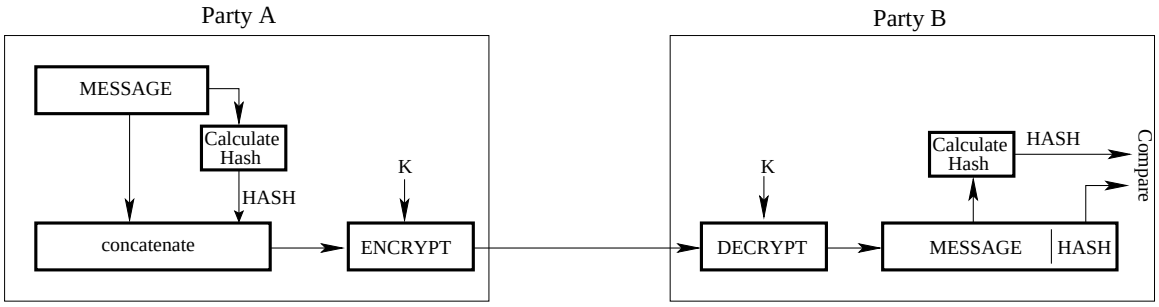
- The scheme in Figure 1(c) is a public-key encryption version of the scheme shown in Figure 1(b). The hashcode of the message is encrypted with the sender's private key. The receiver can recover the hashcode with the sender's public key and authenticate the message as indeed coming from the alleged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

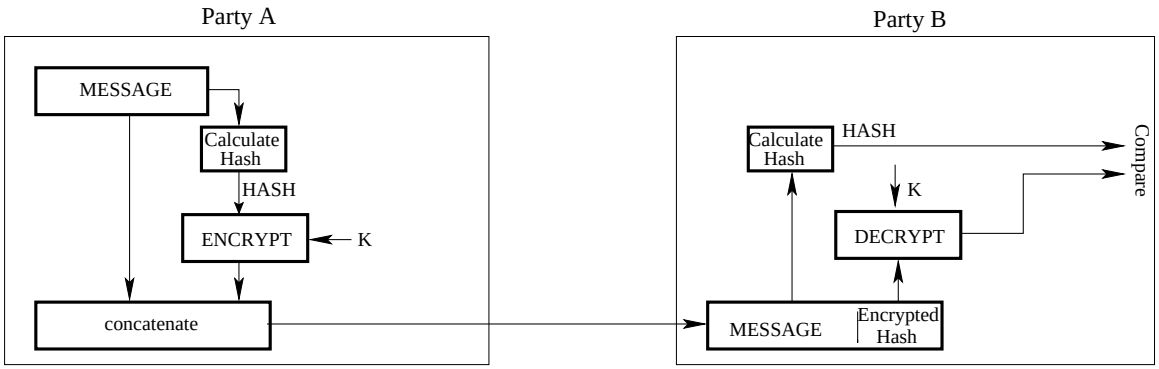
- A very different approach to the use of hashing for authentication is shown in Figure 2(b). In this scheme, nothing is encrypted. However, the sender appends a secret string S, known also to the receiver, to the message before computing its hashcode. Before

checking the hashcode 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 hashcode.

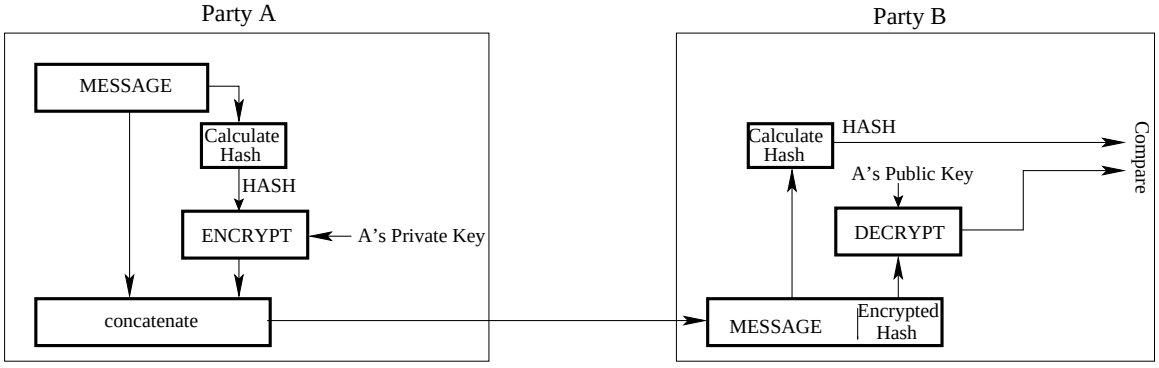
- Finally, the scheme in Figure 2(c) shows an extension of the scheme of Figure 2(b) where we have added symmetric-key based confidentiality to the transmission between the sender and the receiver.



(a)

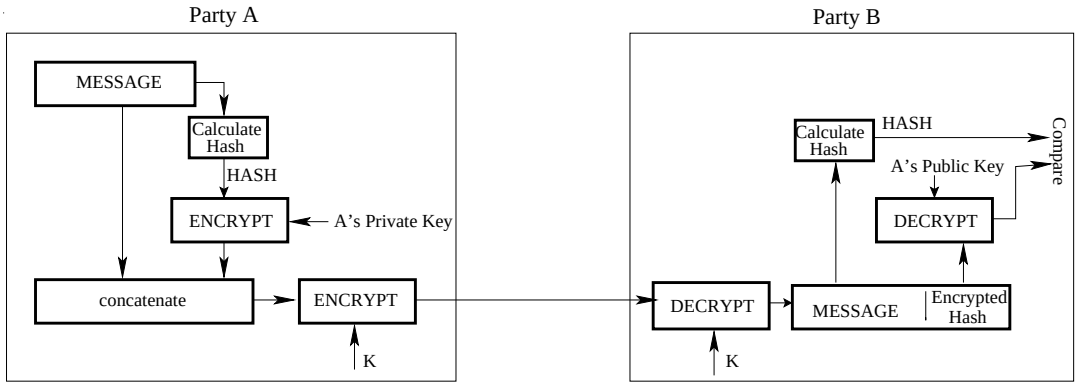


(b)

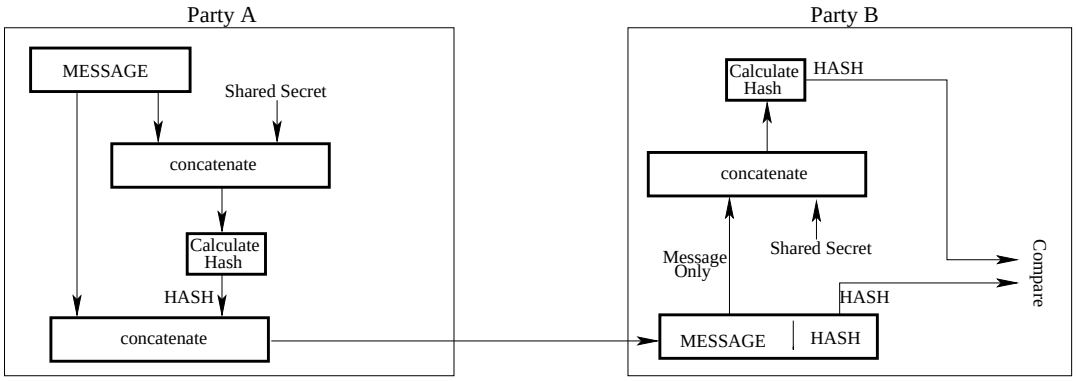


(c)

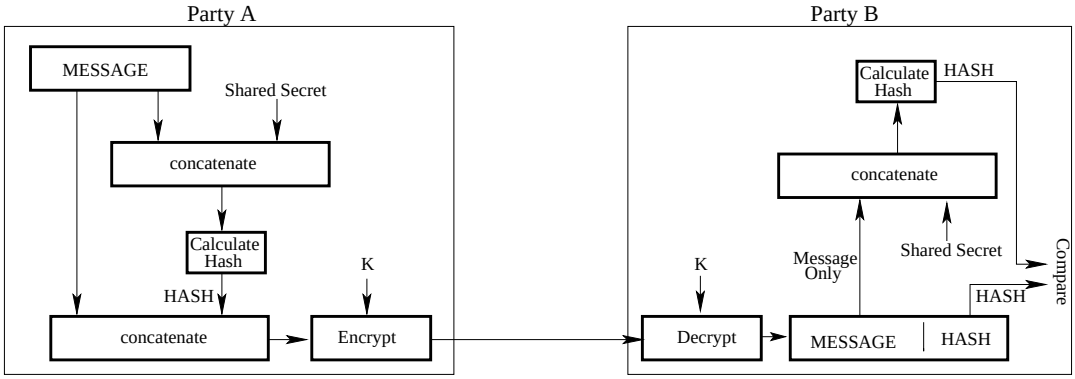
Figure 1: *Different ways of incorporating message hashing in a communication link.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)



(a)



(b)



(c)

Figure 2: *Different ways of incorporating message hashing in a communication link.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

15.3: WHEN IS A HASH FUNCTION CRYPTOGRAPHICALLY SECURE?

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

  - It is **computationally infeasible** to find a message that corresponds to a **given** hashcode. This is sometimes referred to as the **one-way property** of a hash function. [For long messages, that is, messages that are much longer than the length of the hashcode, one may expect this property to hold true trivially. However, note that a hash function **must** possess this property **regardless of the length of the messages**. In other words, it should be just as difficult to recover from its hashcode a message that is as short as, say, a single byte as a message that consists of millions of bytes.]

  - It is **computationally infeasible** to find two **different messages** that hash to the same hashcode 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 hashcode.

- 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 hashcode, there are only 2^n **distinct** hashcode 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 hashcode. However, 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 and there is almost always some sort of an upper bound on the different types of messages that are possible in any given context.] Collision resistance refers to the likelihood that two different messages possessing certain basic structure so as to be meaningful will result in the same hashcode.

- 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. **Authentication here means that the message has not been altered in any way — that is, it is the authentic original message as produced by its author.** In such applications, we would like to send unencrypted plaintext messages along with their encrypted hashcodes. [That would eliminate the computational overhead of encryption and decryption for the main message content and yet allow for its authentication.] But this would work only if the hashing function has perfect collision resistance. [If a hashing approach has poor collision resistance, an adversary could compute the hashcode of the message content and replace it with some other content that has the same hashcode value.]

15.4: SIMPLE HASH FUNCTIONS

- Practically all algorithms for computing the hashcode 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 in order to generate its hashcode.

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

- The hashcode 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 hashcode so that the final hashcode remains unchanged. To see

this more clearly, let $\{X_1, X_2, \ldots, \}$ be the bit blocks of a message M, each block of size n bits. That is $M = (X_1||X_2|| \ldots ||X_m)$. (The operator '||' means concatenation.) The hashcode produced by the XOR algorithm can be expressed as

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

where $\Delta(M)$ is the hashcode. Let's say that an adversary can observe $\{M, \Delta(M)\}$. An adversary can easily create 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 an Y_m that is given by

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

On account of the properties of the XOR operator, it is easy to show that the hashcode for $M_{forged} = \{Y_1||Y_2|| \cdots ||Y_m\}$ will be the same as $\Delta(M)$. Therefore, when the forged message is concatenated with the original $\Delta(M)$, the recipient would not suspect any foul play.

- When you are hashing regular text and the character encoding is based on ASCII (or its variants), the collision resistance property of the XOR algorithm suffers even more because the highest bit in every byte will be zero. Ideally, one would hope that, with an N-bit hashcode, any particular message would result in a given hashcode value with a probability of $\frac{1}{2^N}$. But when the highest bit in each byte for each character is always 0, some of the N bits in the hashcode will predictably be 0 with the simple XOR algorithm. **This obviously reduces the number of unique**

hashcode values available to us, and thus increases the probability of collisions.

- To increase the space of distinct hashcode values available for the different messages, a variation on the basic XOR algorithm consists of performing a one-bit circular shift of the partial hashcode 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 hashcode value h_1; replace M_1 by a message M_2 of hashcode value h_2; append a block of gibberish at the end M_2 to force the hashcode 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 how the length of a message is included in what gets hashed, here is how the now-not-so-popular SHA-1 algorithm pads a message before it is hashed:

```
The very first step in the SHA-1 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) mod 512   =   448

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  ------------------------------>
```

- As to why we append a single bit of '1' at the end of the actual message, see Section 15.7.3 where I have described my Python and Perl implementations of the SHA-1 hashing algorithm.

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 hashcode for each message produced by the generator.

- Let's say we are interested in knowing whether any of the messages is going to have its hashcode 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 hashcode is equal to h with probability 0.5?

- To find k, we reason as follows:

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

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

– Since the hashcode 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 hashcode of neither is equal to h are probabilistically independent. That implies that the probability that **none** of two messages has its hashcode equal to h is $(1 - \frac{1}{N})^2$. [Of course, by similar reasoning, the probability that **both** x and y will have their hashcodes 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 hashcodes equal to h is $(1 - \frac{1}{N})^k$.

– Therefore, the probability that **at least one** of the k messages has its hashcode 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 hashcode 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 hashcode 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 hashcodes. 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 hashcode 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 hashcodes being equiprobable, the probability that any given message has its hashcode 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 at a late-night party, the event that this pool has at least one message whose hashcode is h is made up of the event that the first of the two messages has its hashcode equal to h **or** the event that the second of the two messages has its hashcode equal to h. Since the two events are disjunctive, the probability that a pool of two messages has at least one message whose hashcode 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 exceeds N.**

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

- Assuming that a hash algorithm is working perfectly, meaning that it has no biases in its output that may be induced by either the composition of the messages or by the algorithm itself, the goal of this section is to estimate the smallest size of a pool of randomly selected messages so that there exist at least two messages in the pool with the same hashcode with probability 0.5.

- Given a pool of k messages, the question "*What is the probability that there exists at least one message in the pool whose hashcode 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 hashcodes 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 $\frac{(20\times 19)/2}{365} = \frac{190}{365}$. [*Strictly speaking, as you'll see, this calculation is valid only when the class size is very small compared to 365.*] This is referred to as the **birthday paradox** — it is a paradox only in the sense that it seems counterintuitive. [A quick way to accept the 'paradox' intuitively is that for '20 choose 2' 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%. (These statements are based on the more precise formulas shown in this section.) [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 hashcode value from N possible such values, the probability that the pool will contain at least one pair of messages with the same hashcode 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 hashcodes 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 hashcode.

  – 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 hashcodes in the pool.*

  – For the first message in the pool, we can choose any arbitrarily. Since there exist only N distinct hashcodes, and, therefore, since there can only be N different messages with distinct hashcodes, 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 hashcode, for the second entry in the pool, we can select a message corresponding to the other $N-1$ still available hashcodes.

  – Having used up two distinct hashcode values, for the third entry in the pool, we can select a message corresponding to

the other $N-2$ still available hashcodes; 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 hashcode values is

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

– 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 hashcodes. Reasoning as before, there are N ways to choose the first message. For selecting the second message, we pay no attention to the hashcode 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 hashcode duplication is

$$M_2 \;=\; N \times N \times \ldots \times N \;=\; N^k \qquad (5)$$

– Therefore, if you construct a pool of k purely randomly selected messages, the probability that this pool has no duplications in the hashcodes is

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

– We can now make the following probabilistic inference: if you construct a pool of k message as above, the probability that the pool has *at least* one duplication in the hashcode 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 hashcodes 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

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

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 hashcodes 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

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

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 hashcode can take on a total N different values with equal probability, a pool of $\sqrt{N}$ messages will contain at least one pair of messages with the same hashcode with a probability of 0.5.

- So if we use an n-bit hashcode, 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 hashcode with a probability of 0.5.

- Let's again consider the case of 64 bit hashcodes. Now $N = 2^{64}$. So a pool of 2^{32} randomly generated messages will have at least one pair with identical hashcodes 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 hashcode 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 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 hashcode?*"

- 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 hashcode being any one of N permissible values is $\frac{1}{N}$. Therefore, the probability that the hashcode $h(c_1)$ matches the hashcode $h(f_1)$ is $\frac{1}{N}$. Hence the probability that the hashcode $h(c_1)$ does **not** match the hashcode $h(f_1)$ 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** $\ldots$, $h(f_k)$ 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 hashcode 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 hashcode 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 hashcode, $N = 2^n$. 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 hashcode of the correct version of the contract with his private key, and then replacing the contract by its fraudulent version that has the same hashcode 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 an n-bit hash coding algorithm that has no security flaws, 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

- As previously mentioned in Section 15.3 of this lecture, 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 hashcode would equal a specified value. Additionally, a hash function should be strictly one-way, in the sense that it lets us compute the hashcode for a message, but does not let us figure out a message for a given hashcode — 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.

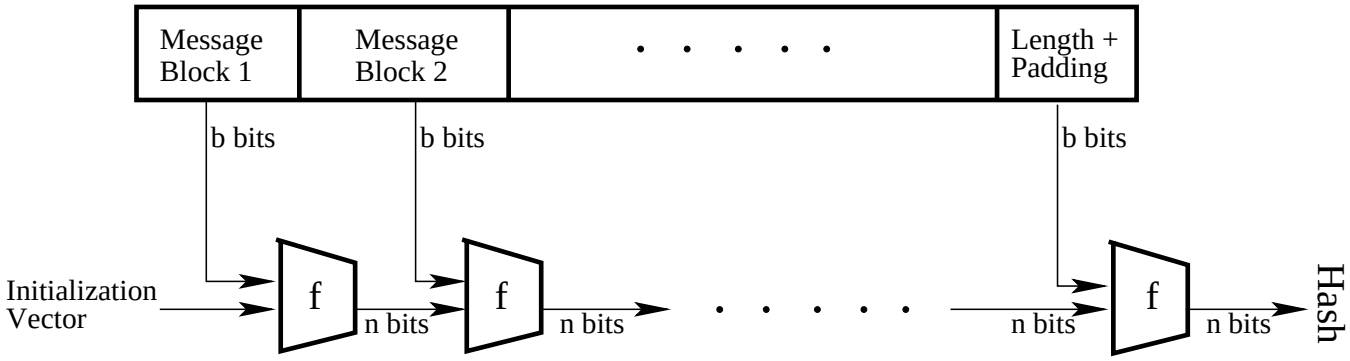


Figure 3: *Merkle's structure for computing a cryptographically secure hash function.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

15.7.1: The SHA Family of Hash Functions

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

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

Algorithm	*Message Size* (bits)	*Block Size* (bits)	*Word Size* (bits)	*Message Digest Size* (bits)	*Security* (ideally) (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 hashcode produced.

– Finally, the *Security* column refers to how many messages would have to be generated before two can be found with the same hashcode with a probability of 0.5 — *assuming that the algorithm has no hidden security holes*. As shown previously in Sections 15.5.1 and 15.6, for a secure hash algorithm **that has no security holes** and that produces n-bit hashcodes, 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 hashcodes.

- SHA-1 was cracked theoretically 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.

- More recently, in February 2017, SHA-1 was actually broken by Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. **They were able to produce two different PDF documents with the same SHA-1 hashcode.** [The title of their paper is "The First Collision For Full SHA-1" and you can download it from `http://shattered.io/`. The attack the authors mounted on SHA-1 is named "The SHAttered attack". The authors say that this attack is 100,000 faster than the brute force attack that relies on the birthday paradox. The authors claim that the brute force attack would require 12,000,000 GPU years to complete, and it is therefore impractical. On the other hand, the SHAttered attack required only 110 years of single-GPU computations. More specifically, according to the authors, the SHAttered attack entailed over 9,223,372,036,854,775,808 SHA1 computations. The authors leveraged the PDF format for creating two different PDFs with the same SHA-1 hash value. To compare SHAttered with the theoretical attack mentioned in the previous bullet, the authors of

SHAttered say their attack took 2^{63} SHA-1 compressions. Note that document formats like PDF that contain macros appear to be particularly vulnerable to attacks like SHAttered. Such documents may lend themselves to what is known as the *chosen-prefix collision attack* in which given two different message prefixes p_1 and p_2, the goal is to find two suffixes s_1 and s_2 so that the hash value for the concatenation $p_1||s_1$ is the same as for the concatenation $p_2||s_2$.]

- 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. Nonetheless, SHA-1 has continued to be widely used in many applications and protocols that require secure and authenticated communications. Unfortunately, SHA-1 continues to be widely used in SSL/TLS, PGP, SSH, S/MIME, and IPSec. (*These standards will be briefly reviewed in Lecture 20.*) Hopefully, going forward, that will stop being the case in light of the real collisions obtained by the SHAttered attack.

- 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. Here is a 4-step summary of SHA-512 processing:

STEP 1: Pad the message so that its length is an integral multiple of 1024 bits, the block size. The only complication here is that the last 128 bits of the last block must contain a value that is the length of the message.

STEP 2: Generate the message schedule required for processing a 1024-bit block of the input message. The message schedule consists of 80 64-bit **words**. The first 16 of these words are obtained directly from the 1024-bit message block. The rest of the words are obtained by applying permutation and mixing operations to the some of the previously generated words.

STEP 3: Apply round-based processing to each 1024-bit input message block. There are 80 rounds to be carried out for each message block. For this round-based processing, we first store the hash values calculated for the PREVIOUS MESSAGE BLOCK in temporary 64-bit variables denoted a, b, c, d, e, f, g, h. In the i^{th} round, we permute the values stored in these eight variables and, with two of the variables, we mix in the message schedule word $words[i]$ and a round constant $K[i]$.

STEP 4: We update the hash values calculated for the PREVIOUS message block by adding to it the values in the temporary variables a, b, c, d, e, f, g, h.

The rest of this section presents in greater detail the different processing steps of SHA-512:

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**.

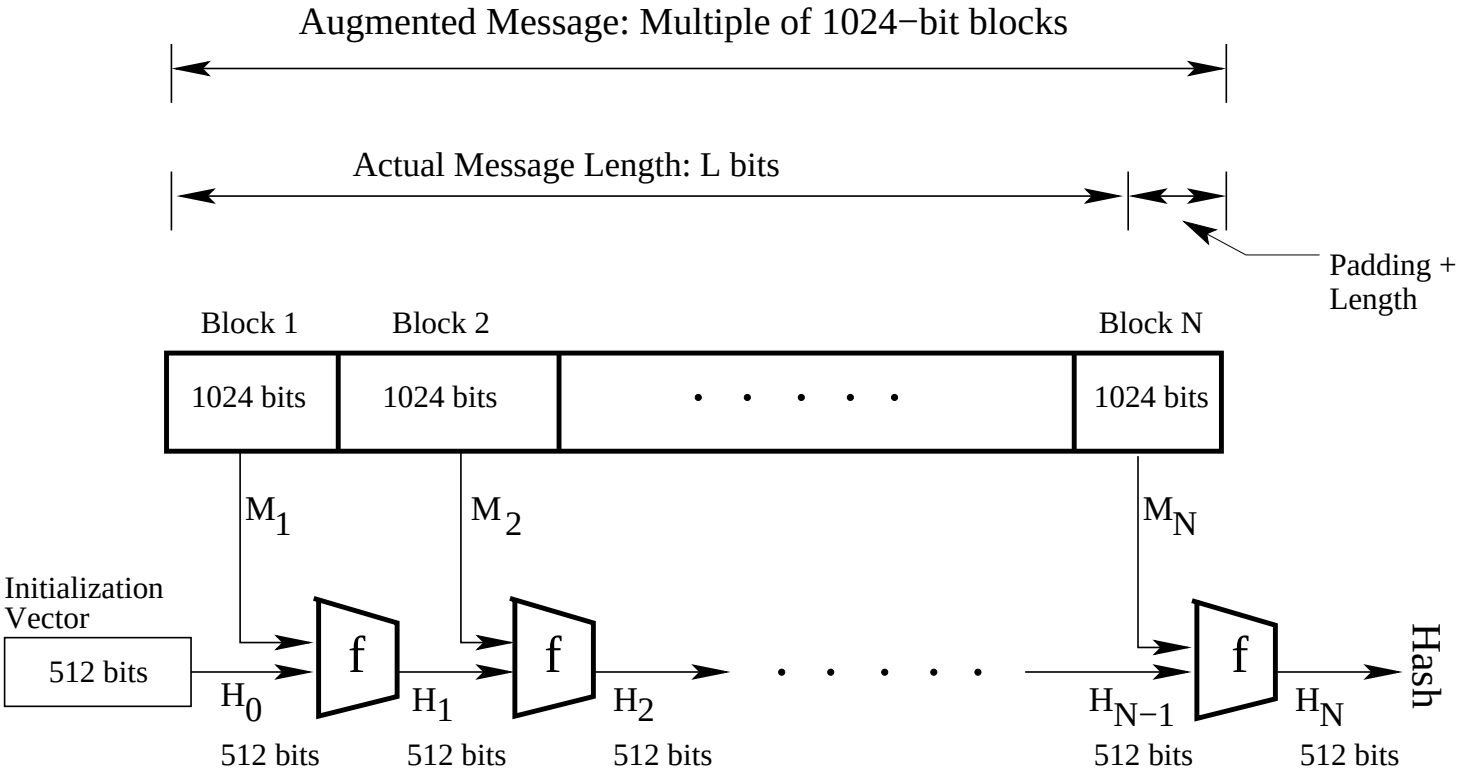


Figure 4: *Overall processing steps of the SHA-512 Secure Hash Algorithm.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

- 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.

- However, before you can carry out round-based processing of each 1024-bit input message block, you must generate what is known as a message schedule.

  The message schedule for SHA-512 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

$$\sigma_0(x) \quad = \quad ROTR^1(x) \; \oplus \; ROTR^8(x) \; \oplus \; SHR^7(x)$$

$$\sigma_1(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)$$

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

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

- As you will see later, in the round-based processing of each input message block, 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 mentioned above 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 row, 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
```

- 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.

- 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

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

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.

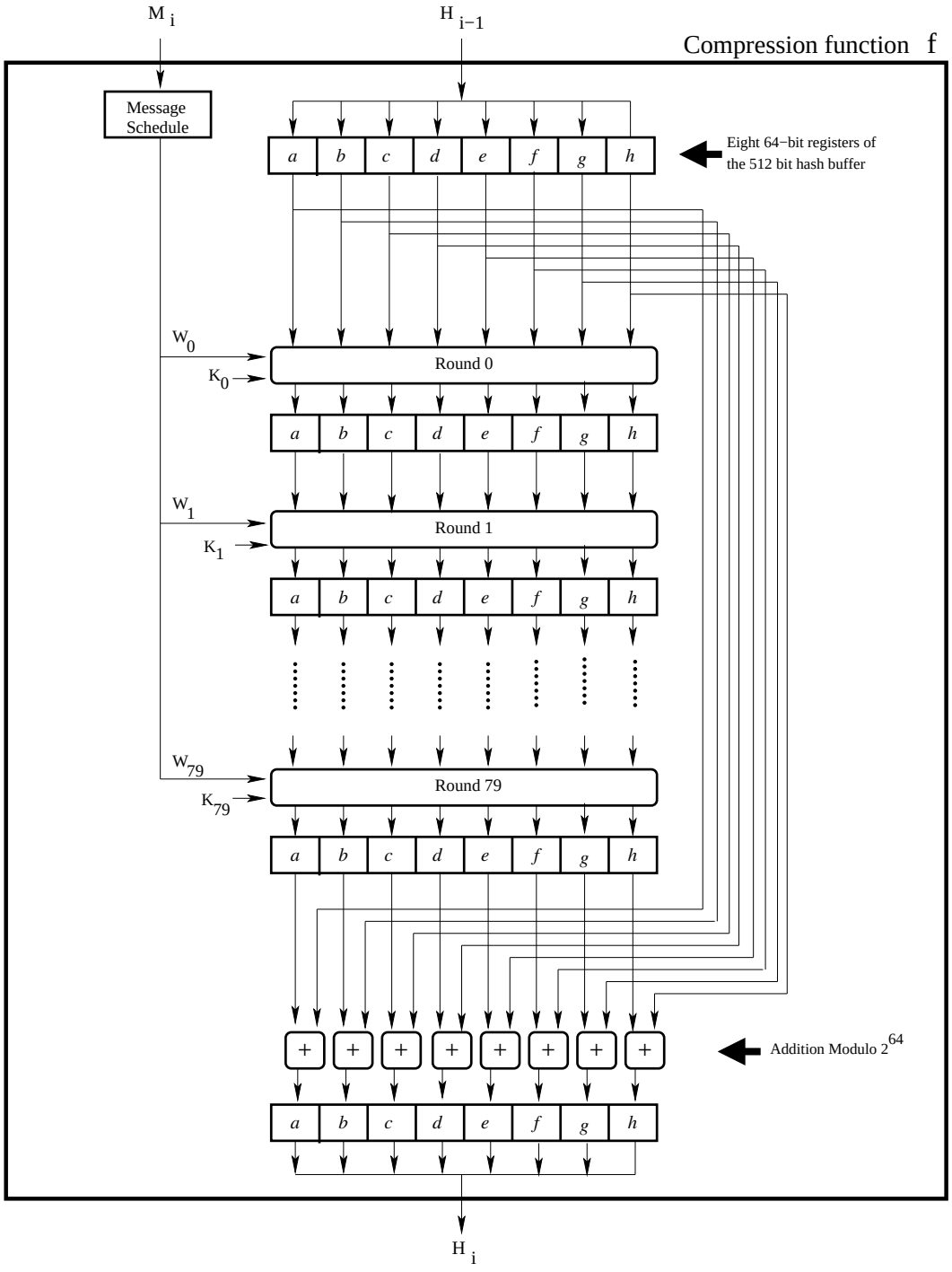


Figure 5: *The 80 rounds of processing that each 1024-bit message block goes through are depicted here.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

15.7.3: Compact Python and Perl Implementations for SHA-1 Using BitVector

- As mentioned in Section 15.7.1, SHA-1 is now to be considered as a completely broken hash function in light of the collision results obtained by the SHAttered attack.

- Despite its having been broken, SHA-1 can still serve as a useful stepping stone if you are learning how to write code for Merkle type hash functions. My goal in this section is to demonstrate my Python and Perl implementations 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 how you can use my BitVector modules (`Algorithm::BitVector` in Perl and `BitVector` in Python) to create compact implementations for cryptographically secure hash algorithms. Typical implementations of the SHA algorithms consist of several hundred lines of code. With `BitVector` in Python and `Algorithm::BitVector` in Perl, you can do the same with a couple of dozen lines.

- 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 and Perl implementations that follow.

- 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 a modulo 512 division of its left operand to return a *nonnegative* remainder less than the modulus 512. 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 bit set to 1.]

- 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 a new 512-bit 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, 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. [This comment about b may appear silly here since the variable b is assigned the value stored in a. However, as shown in the next bullet, at the end 80 rounds of processing, the value of b is used directly for the hashcode for the current 512-bit input block.]

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

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

$$h3 = (h3 + d) \; mod \; 2^{32}$$
$$h4 = (h4 + e) \; mod \; 2^{32}$$

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 implementations shown below are meant to be invoked in a command-line mode as follows:

```
sha1_from_command_line.py    string_whose_hash_you_want

sha1_from_command_line.pl    string_whose_hash_you_want
```

- Here is the Python implementation:

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

##  sha1_from_command_line.py
##  by Avi Kak (kak@purdue.edu)
##  February 19, 2013
##  Modified: March 2, 2016

## Call syntax:
##
##     sha1_from_command_line.py   your_message_string

##  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])) & 0xFFFFFFFF, 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)) & 0xFFFFFFFF, size=32 )
    h1 = BitVector( intVal = (int(h1) + int(b)) & 0xFFFFFFFF, size=32 )
    h2 = BitVector( intVal = (int(h2) + int(c)) & 0xFFFFFFFF, size=32 )
    h3 = BitVector( intVal = (int(h3) + int(d)) & 0xFFFFFFFF, size=32 )
    h4 = BitVector( intVal = (int(h4) + int(e)) & 0xFFFFFFFF, size=32 )

message_hash = h0 + h1 + h2 + h3 + h4
hash_hex_string = message_hash.getHexStringFromBitVector()
sys.stdout.writelines((hash_hex_string, "\n"))
```

- Here are some hash values produced by the above script:

```
sha1_from_command_line.py 0                 =>        b6589fc6ab0dc82cf12099d1c2d40ab994e8410c

sha1_from_command_line.py 1                 =>        356a192b7913b04c54574d18c28d46e6395428ab

sha1_from_command_line.py hello             =>        aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

sha1_from_command_line.py 1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  =>

                                                      475f6511376a8cf1cc62fa56efb29c2ed582fe18
```

- Shown below is the Perl version of the script:

```
#!/usr/bin/env perl

##  sha1_from_command_line.pl
##  by Avi Kak (kak@purdue.edu)
##  March 2, 2016

## Call syntax:
##
##     sha1_from_command_line.pl   your_message_string

##  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.

use strict;
use warnings;
use Algorithm::BitVector 1.25;

die "Usage: %s  <string to be hashed>\n" if @ARGV != 1;

my $message = shift;

# 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.
my $h0 = Algorithm::BitVector->new(hexstring => '67452301');
my $h1 = Algorithm::BitVector->new(hexstring => 'efcdab89');
my $h2 = Algorithm::BitVector->new(hexstring => '98badcfe');
my $h3 = Algorithm::BitVector->new(hexstring => '10325476');
my $h4 = Algorithm::BitVector->new(hexstring => 'c3d2e1f0');

my $bv = Algorithm::BitVector->new(textstring => $message);
my $length = $bv->length();
my $bv1 = $bv + Algorithm::BitVector->new(bitstring => "1");
my $length1 = $bv1->length();
my $howmanyzeros = (448 - $length1) % 512;
my @zerolist = (0) x $howmanyzeros;
my $bv2 = $bv1 + Algorithm::BitVector->new(bitlist => \@zerolist);
my $bv3 = Algorithm::BitVector->new(intVal => $length, size => 64);
my $bv4 = $bv2 + $bv3;

my @words = (undef) x 80;
my @words_bv = (undef) x 80;

for (my $n = 0; $n < $bv4->length(); $n += 512) {
    my @block = @{$bv4->get_bit( [$n .. $n + 511] )};
    @words = map {[@block[$_ * 32 .. ($_ * 32 + 31)]]}  0 .. 15;
    @words_bv = map {Algorithm::BitVector->new( bitlist => $words[$_] )} 0 .. 15;

    my ($a,$b,$c,$d,$e) = ($h0,$h1,$h2,$h3,$h4);
    my ($f,$k);
    foreach my $i (16 .. 79) {
        $words_bv[$i] = $words_bv[$i-3] ^ $words_bv[$i-8] ^ $words_bv[$i-14] ^ $words_bv[$i-16];
        $words_bv[$i] = $words_bv[$i] << 1;
    }
    foreach my $i (0 .. 79) {
        if (($i >= 0) && ($i <= 19)) {
            $f = ($b & $c) ^ ((~$b) & $d);
            $k = 0x5a827999;
        } elsif (($i >= 20) && ($i <= 39)) {
            $f = $b ^ $c ^ $d;
            $k = 0x6ed9eba1;
        } elsif (($i >= 40) && ($i <= 59)) {
            $f = ($b & $c) ^ ($b & $d) ^ ($c & $d);
            $k = 0x8f1bbcdc;
        } elsif (($i >= 60) && ($i <= 79)) {
```

```
            $f = $b ^ $c ^ $d;
            $k = 0xca62c1d6;
        }
        my $a_copy = $a->deep_copy();
        my $T = Algorithm::BitVector->new( intVal => (int($a_copy << 5) + int($f)
                        + int($e) + int($k) + int($words_bv[$i])) & 0xFFFFFFFF, size => 32 );
        $e = $d;
        $d = $c;
        my $b_copy = $b->deep_copy();
        $b_copy = $b_copy << 30;
        $c = $b_copy;
        $b = $a;
        $a = $T;
    }
    $h0 = Algorithm::BitVector->new( intVal => (int($h0) + int($a)) & 0xFFFFFFFF, size => 32 );
    $h1 = Algorithm::BitVector->new( intVal => (int($h1) + int($b)) & 0xFFFFFFFF, size => 32 );
    $h2 = Algorithm::BitVector->new( intVal => (int($h2) + int($c)) & 0xFFFFFFFF, size => 32 );
    $h3 = Algorithm::BitVector->new( intVal => (int($h3) + int($d)) & 0xFFFFFFFF, size => 32 );
    $h4 = Algorithm::BitVector->new( intVal => (int($h4) + int($e)) & 0xFFFFFFFF, size => 32 );
}

my $message_hash = $h0 + $h1 + $h2 + $h3 + $h4;
my $hash_hex_string = $message_hash->get_hex_string_from_bitvector();
print "$hash_hex_string\n";
```

- As you would expect, this script produces the same hash values as the Python version shown earlier in this section:

```
sha1_from_command_line.pl 0                 =>        b6589fc6ab0dc82cf12099d1c2d40ab994e8410c

sha1_from_command_line.pl 1                 =>        356a192b7913b04c54574d18c28d46e6395428ab

sha1_from_command_line.pl hello             =>        aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

sha1_from_command_line.pl 1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  =>

                                                      475f6511376a8cf1cc62fa56efb29c2ed582fe18
```

15.7.4: Compact Python Implementation for SHA-256 Using BitVector

- SHA-256 uses a message block size of 512 bits and, as you'd expect from its name, produces a message digest of size 256 bits. [For comparison, recall that SHA-1 was also based on 512-bit message blocks, but that the message digest it produces is of size 160 bits. On the other hand, SHA-512 uses 1024-bit message blocks and produces 512-bit message digests.]

- The logic of SHA-256 is very similar to that of SHA-512 that I presented in Section 15.7.2 of this lecture. Obviously, as you would expect, all the constants involved are different.

- In the rest of this section, I'll just summarize the main algorithmic steps in SHA-256 and then show its Python implementation using the `BitVector` module.

- SHA256 algorithm in summary:

  STEP 1: Pad the message so that its length is an integral multiple of 512 bits, the block size. The only complication here is that the last 64 bits of the last block must contain a value that is the length of the message.

STEP 2: Generate the MESSAGE SCHEDULE required for processing a 512-bit block of the input message. The message schedule consists of 64 32-bit **words**. The first 16 of these words are obtained directly from the 512-bit message block. The rest of the words are obtained by applying permutation and mixing operations to the some of the previously generated words.

STEP 3: Apply round-based processing to each 512-bit input message block. There are 64 rounds to be carried out for each block. For this round-based processing, we first store the hash values calculated for the PREVIOUS MESSAGE BLOCK in temporary 32-bit variables denoted a, b, c, d, e, f, g, h. In the i^{th} round, we permute the values stored in these eight variables and, with two of the variables, we mix in the message schedule word $words[i]$ and a round constant $K[i]$.

STEP 4: We update the hash values calculated for the PREVIOUS message block by adding to it the values in the temporary variables a, b, c, d, e, f, g, h.

- Shown below is the Python implementation of SHA-256. If you wish, you can download it from the lecture notes website. The website contains the following two different versions of this code:

```
sha256_file_based.py          (which is what you see below)

sha256_from_command_line.py
```

The former is intended for hashing the contents of a text file that you supply to the script as its command-line argument and the latter intended to directly hash whatever string you place in the command-line after the name of the script.

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

##  sha256_file_based.py
##  by Avi Kak (kak@purdue.edu)
##  January 3, 2018

##  Call syntax:
##
##     sha256_file_based.py   your_file_name

##  The above command line applies the SHA256 algorithm implemented here to the contents of the
##  named file and prints out the hash value for the file at its standard output.  NOTE: IT
##  ADDS A NEWLINE 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  <the name of the file>\n" % sys.argv[0])
    sys.exit(1)

with open(sys.argv[1], 'r') as file_to_hash:
    message = file_to_hash.read()

#  The 8 32-words used for initializing the 512-bit hash buffer before we start scanning the
#  input message block for its hashing. See page 13 (page 17 of the PDF) of the NIST standard.
#  Note that the hash buffer consists of 8 32-bit words named h0, h1, h2, h3, h4, h5, h6, and h7.
h0 = BitVector(hexstring='6a09e667')
h1 = BitVector(hexstring='bb67ae85')
h2 = BitVector(hexstring='3c6ef372')
h3 = BitVector(hexstring='a54ff53a')
h4 = BitVector(hexstring='510e527f')
h5 = BitVector(hexstring='9b05688c')
h6 = BitVector(hexstring='1f83d9ab')
h7 = BitVector(hexstring='5be0cd19')

#  The K constants (also referred to as the "round constants") are used in round-based processing of
#  each 512-bit input message block.  There is a 32-bit constant for each of the 64 rounds. These are
#  as provided on page 10 (page 14 of the PDF) of the NIST standard.  Note that these are ONLY USED
#  in STEP 3 of the hashing algorithm where we take each 512-bit input message block through 64
#  rounds of processing.
K = ["428a2f98", "71374491", "b5c0fbcf", "e9b5dba5", "3956c25b", "59f111f1", "923f82a4", "ab1c5ed5",
     "d807aa98", "12835b01", "243185be", "550c7dc3", "72be5d74", "80deb1fe", "9bdc06a7", "c19bf174",
     "e49b69c1", "efbe4786", "0fc19dc6", "240ca1cc", "2de92c6f", "4a7484aa", "5cb0a9dc", "76f988da",
     "983e5152", "a831c66d", "b00327c8", "bf597fc7", "c6e00bf3", "d5a79147", "06ca6351", "14292967",
     "27b70a85", "2e1b2138", "4d2c6dfc", "53380d13", "650a7354", "766a0abb", "81c2c92e", "92722c85",
     "a2bfe8a1", "a81a664b", "c24b8b70", "c76c51a3", "d192e819", "d6990624", "f40e3585", "106aa070",
     "19a4c116", "1e376c08", "2748774c", "34b0bcb5", "391c0cb3", "4ed8aa4a", "5b9cca4f", "682e6ff3",
     "748f82ee", "78a5636f", "84c87814", "8cc70208", "90befffa", "a4506ceb", "bef9a3f7", "c67178f2"]
```

```
#  Store the 64 K constants as an array of BitVector objects:
K_bv = [BitVector(hexstring = k_constant) for k_constant in K]


#  STEP 1 OF THE HASHING ALGORITHM: Pad the input message so that its length is an integer multiple
#                                   of the block size which is 512 bits.  This padding must account
#                                   for the fact that the last 64 bit of the padded input must store
#                                   length of the input message:
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

#  Initialize the array of "words" for storing the message schedule for each block of the
#  input message:
words = [None] * 64

for n in range(0,bv4.length(),512):
    block = bv4[n:n+512]

    #  STEP 2 OF THE HASHING ALGORITHM: Now we need to create a message schedule for this 512-bit
    #                                   input block.  The message schedule contains 64 words, each
    #                                   32-bits long.  As shown below, the first 16 words of the
    #                                   message schedule are obtained directly from the 512-bit
    #                                   input block:
    words[0:16] = [block[i:i+32] for i in range(0,512,32)]
    #  Now we need to expand the first 16 32-bit words of the message schedule into a full schedule
    #  that contains 64 32-bit words. This involves using the functions sigma0 and sigma1 as shown
    #  below:
    for i in range(16, 64):
        i_minus_2_word = words[i-2]
        i_minus_15_word = words[i-15]
        #  The sigma1 function is applied to the i_minus_2_word and the sigma0 function is applied
        #  to the i_minus_15_word:
        sigma0 = (i_minus_15_word.deep_copy() >> 7) ^ (i_minus_15_word.deep_copy() >> 18) ^ \
                                                      (i_minus_15_word.deep_copy().shift_right(3))
        sigma1 = (i_minus_2_word.deep_copy() >> 17) ^ (i_minus_2_word.deep_copy() >> 19) ^ \
                                                      (i_minus_2_word.deep_copy().shift_right(10))
        words[i] = BitVector(intVal=(int(words[i-16]) + int(sigma1) + int(words[i-7]) +
                                                              int(sigma0)) & 0xFFFFFFFF, size=32)

    #  Before we can start STEP 3, we need to store the hash buffer contents obtained from the
    #  previous input message block in the variables a,b,c,d,e,f,g,h:
    a,b,c,d,e,f,g,h = h0,h1,h2,h3,h4,h5,h6,h7

    #  STEP 3 OF THE HASHING ALGORITHM: In this step, we carry out a round-based processing of
    #                                   each 512-bit input message block.  There are a total of
    #                                   64 rounds and the calculations carried out in each round
    #                                   are referred to as calculating a "round function".  The
```

```
        #                                   round function for the i-th round consists of permuting
        #                                   the previously calculated contents of the hash buffer
        #                                   registers as stored in the temporary variables
        #                                   a,b,c,d,e,f,g and replacing the values of two of these
        #                                   variables with values that depend of the i-th word in the
        #                                   message schedule, words[i],  and i-th round constant, K[i].
        #                                   As you see below, this requires that we first calculate the
        #                                   functions ch, maj, sum_a, and sum_e.
        for i in range(64):
            ch = (e & f) ^ ((~e) & g)
            maj = (a & b) ^ (a & c) ^ (b & c)
            sum_a = ((a.deep_copy()) >> 2) ^ ((a.deep_copy()) >> 13) ^ ((a.deep_copy()) >> 22)
            sum_e = ((e.deep_copy()) >> 6) ^ ((e.deep_copy()) >> 11) ^ ((e.deep_copy()) >> 25)
            t1 = BitVector(intVal=(int(h) + int(ch) + int(sum_e) + int(words[i]) +
                                                              int(K_bv[i])) & 0xFFFFFFFF, size=32)
            t2 = BitVector(intVal=(int(sum_a) + int(maj)) & 0xFFFFFFFF, size=32)
            h = g
            g = f
            f = e
            e = BitVector(intVal=(int(d) + int(t1)) & 0xFFFFFFFF, size=32)
            d = c
            c = b
            b = a
            a = BitVector(intVal=(int(t1) + int(t2)) & 0xFFFFFFFF, size=32)

        #  STEP 4 OF THE HASHING ALGORITHM:  The values in the temporary variables a,b,c,d,e,f,g,h
        #                                    AFTER 64 rounds of processing are now mixed with the
        #                                    contents of the hash buffer as calculated for the previous
        #                                    block of the input message:
        h0 = BitVector( intVal = (int(h0) + int(a)) & 0xFFFFFFFF, size=32 )
        h1 = BitVector( intVal = (int(h1) + int(b)) & 0xFFFFFFFF, size=32 )
        h2 = BitVector( intVal = (int(h2) + int(c)) & 0xFFFFFFFF, size=32 )
        h3 = BitVector( intVal = (int(h3) + int(d)) & 0xFFFFFFFF, size=32 )
        h4 = BitVector( intVal = (int(h4) + int(e)) & 0xFFFFFFFF, size=32 )
        h5 = BitVector( intVal = (int(h5) + int(f)) & 0xFFFFFFFF, size=32 )
        h6 = BitVector( intVal = (int(h6) + int(g)) & 0xFFFFFFFF, size=32 )
        h7 = BitVector( intVal = (int(h7) + int(h)) & 0xFFFFFFFF, size=32 )

    #  Concatenate the contents of the hash buffer to obtain a 512-element BitVector object:
    message_hash = h0 + h1 + h2 + h3 + h4 + h5 + h6 + h7

    #  Get the hex representation of the binary hash value:
    hash_hex_string = message_hash.getHexStringFromBitVector()

    sys.stdout.writelines((hash_hex_string, "\n"))
```

15.8: HASH FUNCTIONS FOR COMPUTING MESSAGE AUTHENTICATION CODES

- As you saw in Section 15.2 (and through the different modes of using a hash function shown in Figure 1 in that section), in computer security work, a hash value for a message is of no use unless it is encrypted in some form. Sending a message along with an unencrypted hash provides zero security against forgery. A **message authentication code (MAC)** is just a slight variation on what you already know about combining a hash with an element of secrecy in some form. You might say that MAC is trade jargon for how exactly a hash is used with secrecy rather than a fundamentally new concept. As long as we are talking about trade jargon, 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 hashcode is one way to generate the MAC of a message. This is the same thing as shown earlier in Figure 2(b).** A MAC produced in this manner is sometimes called **HMAC** where the letter 'H' stands for "Hash." The encryption that must be applied to the hash can be based on a block cipher or a stream

cipher. A block-cipher based MAC called **DES-CBC MAC** is used in various standards.

- Other 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 hashcode, the new composite hashed again, and so on.

- 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 the following encrypted checksum as the MAC value for the message M:

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

  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 create 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. **This is exactly the same point that was made earlier in Section 15.4.**

- 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.

- I alluded to **HMAC** earlier in this section as a form of MAC. HMAC is used in the IPSec protocol (for packet-level security in computer networks), in SSL (for transport-level security), and

other applications. We will discuss these protocols in Lecture 20. The size of the MAC produced by HMAC is the same as the size of the hashcode 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.

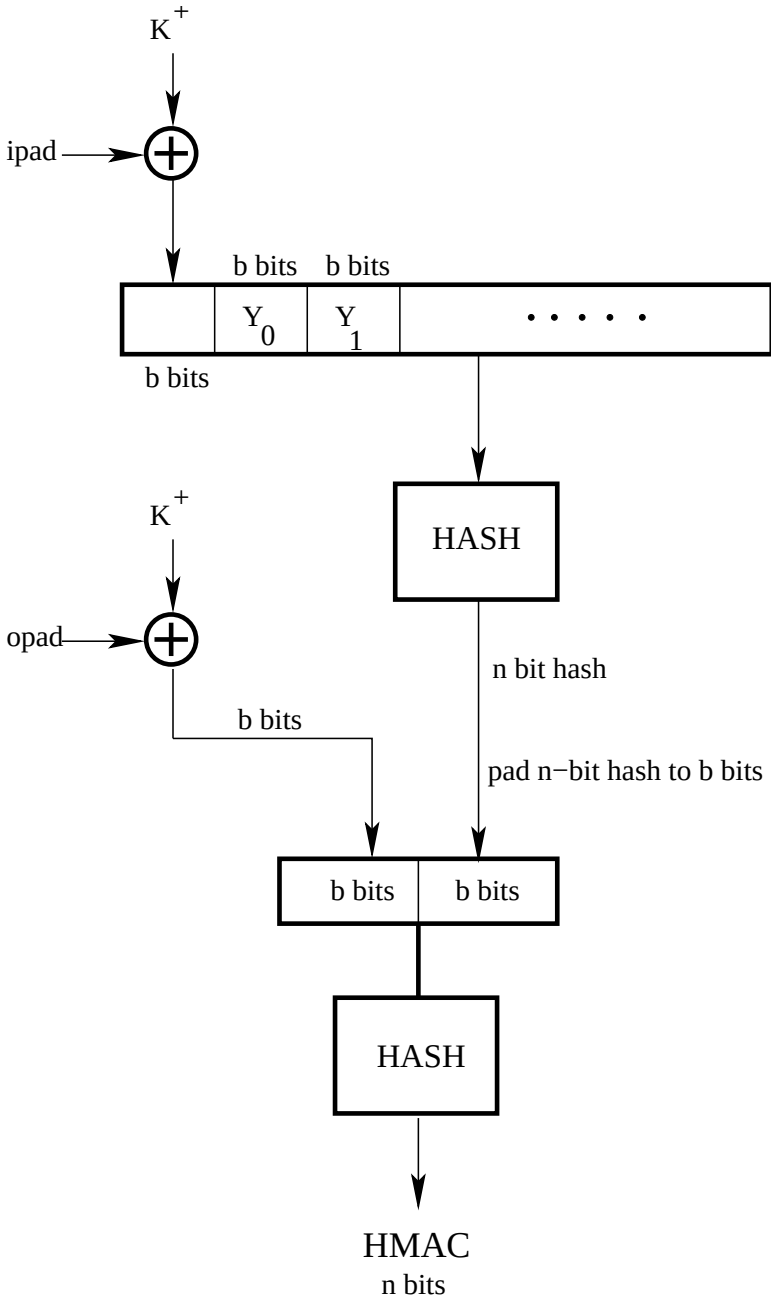


Figure 6: *Operation of the HMAC algorithm for computing a message authentication code.* (*This figure is from "Computer and Network Security" by Avi Kak*)

15.9: CRYPTO CURRENCIES AND THEIR USE OF HASHING

- Hashing finds extensive use in crypto currency algorithms. Hashing algorithms are used **(1)** in what is known as "proof-of-work" calculations for "mining" a new coin; **(2)** to authenticate ownership of coins; **(3)** to establish a protocol for transferring the ownership from one node to another in a cryptocoin network; **(4)** and to create a protection against what is known as "**double spending**," which means that no node should be able to sell the same coin to multiple other nodes in the network.

- A node in crypto coin network may establish ownership of a coin by first taking a hash of all the relevant parameters related to the coin and encrypting the hash with its private key. Transferring the ownership of a coin from one node to another in a coin network may be established by the receiving node sending its public key to the selling node; the latter may then append that public key to the coin, take a hash of the resulting structure, digitally encrypt the hash with its private key, and send the resulting object to the buyer. Hashing also plays a critical role in preventing double spending as I'll explain later in this section.

- I'll explain these and other related concepts with the help of the `CeroCoinClient` module in Python that I specifically created for this educational exercise. So I'd encourage the reader to visit the following webpage in order to become familiar with the `CeroCoin` virtual currency:

  `https://engineering.purdue.edu/kak/distCC/CeroCoinClient-2.0.1.html`

- As you will learn from `CeroCoin`, a crypto currency is based on the following basic notions:

  1. Creating rare software objects through what are known as "proof of work" calculations. **These objects becomes the coins of a virtual currency**.

  2. Public key cryptography for digital signatures

  3. The notion of a transaction that comes with a verifiable proof of ownership transfer from the seller of a coin to its buyer.

  4. The notion of a block as a set of transactions and, even more importantly, the notion a blockchain that is used to implicitly embed in each block a hash of all the previous transactions carried out so far. This provides security against what is known as double spending. This is what makes it practically possible to use crypto currencies in a manner similar to real currencies.

- **It is interesting to note that a crypto currency can only exist in a network of nodes in the internet.** This is also true of real

currencies — a currency would lose its meaning if its use was confined to a single individual. [Imagine a lone survivor of a shipwreck stranded on an island with no other inhabitants. Further imagine that the survivor somehow managed to salvage his/her trunk full of gold and diamonds. With on one else on the island to covet his/her wealth, its value would suddenly become no different from that of the sand on the island.]

What is a Coin in a Virtual Currency?:

- Before answering this question, let's first talk about "rare things" and their social value.

- It is not uncommon for things that are rare to acquire enhanced value in our societies. There can be a number of reasons for the high value associated with such objects: their beauty, their history, their cultural significance, the social prestige associated with their possession, etc. [Consider, for example, the Fabergé eggs — highly ornate and jeweled — that were made by the House of Fabergé between 1885 and 1917 for the Russian tsars and other wealthy individuals. Only 65 of these eggs were made and, according to the Wikipedia, 57 have survived. Each egg is now worth several million dollars.]

- Let's now switch over to the virtual world and pose the questions:
  - Is it possible to create rare things in cyberspace?
  - If it is possible to create such objects in cyberspace, is it possible to make them objects of desire?

- Let's ponder the first question first: Is it possible to create rare things that reside only in the memory of a computer? For a software object to be rare, it must satisfy the following two requirements:
  - It must be extremely difficult to produce the object; and
  - it must be extremely easy to verify that it was extremely difficult to produce the object.

- What's interesting is that we can use hash function to satisfy both these computational requirements:

  If we require the rare software object to possess a hash value that has a particular pattern to it, you'd have to mount a large computational effort to discover a message whose hash would correspond to that pattern. After you have found such a message, it would be trivial for anyone to verify your discovery.

- Say we use an N-bit hash function whose output can be thought of as being truly random. Let's say that our requirement for rareness is that the hash value be smaller than some integer t. Considering that an N-bit hash is an integer in the interval $[0, 2^{N-1}]$ and the fact that a good function should distribute the output integers over the entire range uniformly, the probability that you will discover a message whose hash is upper bounded

by t is given by $t/2^N$. This probability can be made as small as you wish by making t sufficiently small for any reasonable value for N. Consider, for example, $t = 1000$ and $N = 256$. Now we are talking about a rareness probability of $1000/2^{256}$, which is a number pretty close to zero.

- Here is how a freshly mined coin looks like in `CeroCoin`:

```
CEROCOIN COIN_ID=e0051e79 CREATOR=e31645f56058f1e450ce17c86752bbc0fe74fa9a69f295323bf793ec7185a7ea
CREATOR_PUB_KEY=CEROCOIN-1.4,mod=cd0b1fa36157757455575d625b6cd2262c11cd50aedc2d89d5a3d3ed590e344a5d8bfcfed0581ddf7836ccda1eb1417b8ea2bc66047fe107c8643575c147c3577,e=10001
MESSAGE_STRING=e671b364d94e7f644c8c170bd6f2d86abee7b10961703d9d5bbcb251a346bbfca5b1ac01dbe476961b0aa70e8f95c4a2de38b7787ada49c5dd4bfd620c246b4316d91a1a8702825f01d1e34332a77a815a66a0e647
c1ef083d2e5292cfc73737b06ac15f41e0fe5a4077bafcb5cd48524cf88b0a7148771280da11d81c3e187a88b1cc2c8808e5e291795e603e092e65629cc2f25eba5cd26ad06aec1a69d850c2ef02e3aa064fdca0d8a16c5118bc9766
b3b31be07340dfb51beb1a9755f99b7b73d53e8f551d6c27af45075632d058aa6f34bf5e86b1639a5d203c94a78970fc16128b7c16774a16cc0744be23c039b48d89d059ca1a2a81964d8c248734483e45a6de4c43d3f9c51fa2caf4
ff9929caaa60380888bbedeb1388b334831afea80621f4aba0c7221b8bc02182b1ac05316d6a80f2005bceca3859a3145ec72fa4c40853c7d3dc4459f5079afe2c9b5a22de314f0bc128463ee0689cca536109c1bdb3f93269a8d02f
3dba188ffa8f72780970bc8faf4a7b5182421a244e38d8979619a7de2e8612aa1522575d907b1be932ac298a0ac496529f16ce8acd6495c58fd2391d1619e8c0b87a802bda71da177ca7b55ed35fdae33bda98a81cee67312151ef092da
2e91d75809b3db3c1a3432c2a430ab61ebc5f297774a880842cf0e43f93464821fc3b2b8dcdc2fc96371bf9df31b83b42e1b5b57014cc1d64f9d6b
NONCE=3e45a6de4c43d3f9c51fa2caf4ff9929caaa6038088bbedeb1388b334831afea80621f4aba0c7221b8bc02182b1ac05316d6a80f2005bceca3859a3145ec72fa4c40853c7d3dc4459f5079afe2c9b5a22de314f0bc128463ee0689cca536109c
POW_DIFFICULTY_LEVEL=252 TIMESTAMP=1519751894.29 MSG_HASH=0ce65c1874662ba9d64755a983db9056003a6065c80d068c4a1459df50489b6d
```

- If you blow up the coin printout shown above, you will notice the following entry in the last line:

```
POW_DIFFICULTY_LEVEL=252
```

in which the prefix "POW" stands for "Proof of Work", which is the level of difficulty that was used for generating the coin. Obviously, this difficulty level is meant to establish rarity of the coin. Setting `POW_DIFFICULTY_LEVEL` to 252 means that a message hash would be considered acceptable for discovering a new coin provided its value is **less than** 2^{252}. In the same line where you see the above entry, you will also see the following entry for the message hash:

```
MSG_HASH=0ce65c1874662ba9d64755a983db9056003a6065c80d068c4a1459df50489b6d
```

Note that the leading character in the hex representation of the 256-bit hash is "0", which means that the first four bits of the

message hash are all zeros, which is exactly what is required by the condition on the "Proof-of-Work" difficulty level. [**The hash value was produced by the SHA256 algorithm whose BitVector based implementation comes with the CeroCoinClient module.**]

- **The value of 252 for `POW_DIFFICULTY_LEVEL` would be much too silly for discovering coins that would command any value in the marketplace of crypto currencies. However, this value is convenient if you want to show coin production during the time limits imposed by a classroom demonstration of `CeroCoin`.**

- Another criterion for rareness could be requiring that a certain designated number of the least significant bits of the hash value be all zeros.

How to Establish Ownership of a Coin?:

- Now that we know how to create rare objects in the virtual world, how does one establish ownership for them? And, assuming we can come up with a foolproof method for solving the ownership question, how does one transfer the owner to another user?

- Even before we take up the question of establishing ownership, we need to figure out how to establish virtual-world identities for the human owners of the rare objects in the virtual world.

- A human user's identity (ID) in the virtual world can be the hash of his/her public key. The public key being after all public would be available to all who wish to verify that ID. For someone to verify an ID, all they have to do is to hash the public key of the individual and see if it yields the same value as the ID under examination.

- With the issue of establishing identity settled, let us next address the issue of how to establish ownership of a coin. With public-key crypto, that is easy to do. All the coin discoverer has to do is to take hash of the coin, encrypt it with his/her private key, and append the signature to the coin.

- After we have instantiated a basic coin, the coin creator must sign it with his/her private key. This is done by the method `digitally_sign_the_coin()` of the `CeroCoinClient` class shown below. This method first applies the SHA256 hash function to the string representation of the coin. The call to `SHA256.SHA256()` returns the hash as a string of hex characters that we turn into an integer. Subsequently, we exponentiate this integer with the private key of the coin creator. Finally, we append the signature to the coin.

```
def digitally_sign_coin(self, coin):
    mdebug = True
    modulus,e,d,p,q,Xp,Xq = self.modulus,self.pub_exponent,self.priv_exponent,self.p,self.q,self.Xp,self.Xq
    if debug:
        print("\nDIGI_SIGN -- modulus: %d" % modulus)
        print("\nDIGI_SIGN -- public exp: %d" % e)
        print("\nDIGI_SIGN -- private exp: %d" % d)
```

```
            print("\nDIGI_SIGN -- p: %d" % p)
            print("\nDIGI_SIGN -- q: %d" % q)
        splits = str(coin).split()
        coin_without_ends = " ".join(str(coin).split()[1:-1])
        hasher = SHA256.SHA256(message = str(coin_without_ends))
        hashval = hasher.sha256()
        if mdebug:
            print("\nDIGI_SIGN: hashval for coin as int: %d" % int(hashval, 16))
        hashval_int = int(hashval, 16)
        Vp = modular_exp(hashval_int, d, p)
        Vq = modular_exp(hashval_int, d, q)
        coin_signature = (Vp * Xp) % modulus + (Vq * Xq) % modulus
        if mdebug:
            print("\nDIGI_SIGN: coin signature as int: %d" % coin_signature)
        coin_signature_in_hex = "%x" % coin_signature
        coin_with_signature = self.insert_miner_signature_in_coin(str(coin), coin_signature_in_hex)
        checkval_int = modular_exp(coin_signature, e, modulus)
        if mdebug:
            print("\nDIGI_SIGN: coin hashval as int:  %d" % hashval_int)
            print("\nDIGI_SIGN: coin checkval as int: %d" % checkval_int)
        assert hashval_int == checkval_int, "coin hashval does not agree with coin checkval"
        if mdebug:
            print("\nThe coin is authentic since its hashval is equal to its checkval")
        return coin_with_signature
```

- After the miner has signed the coin, this is how it looks like:


```
CEROCOIN COIN_ID=e0051e79 CREATOR=e31645f56058f1e450ce17c86752bbc0fe74fa9a69f295323bf793ec7185a7ea
CREATOR_PUB_KEY=CEROCOIN-1.4,mod=cd0b1fa36157757455755d625b6cd2262c11cd50aedc2d89d5a3d3ed590e344a5d8bfcfed0581ddf7836ccda1eb1417b8ea2bc66047fe107c8643575c147c3577,e=10001
MESSAGE_STRING=e671b364d94e7f644c8c170bd6f2d86abee7b10961703d9d5bbcb251a346bbfca5b1ac01dbe476961b0aa70e8f95c4a2de38b7787ada49c5dd4bfd620c246b4316d91a1a8702825f01d1e34332a77a815a66a0e647
c1ef083d2e5292cfc73737b06ac15f41e0fe5a4077bafcb5cd48524cf88b0a7148771280da11d81c3e187a88b1cc2c8808e5e291795e603e092e65629cc2f25eba5cd26ad06aec1a69d850c2ef02e3aa064fdca0d8a16c5118bc9766b
3b31be07340dfb51beb1a9755f99b7b73d53e8f551d6c27af45075632d058aa6f34bf5e86b1639a5d203c94a78970fc16128b7c16774a16cc0744be23c039b48d89d059ca1a2a8196408c248734483e45a6de4c43d3f9c51fa2caf4ff
9929caaa6038088bbedeb1388b334831afea80621f4aba0c7221b8bc02182b1ac05316d6a80f2005bceca3859a3145ec72fa4c40853c7d3dc4459f5079afe2c9b5a22de314f0bc128463ee0689cca536109c1bdb3f93269a8d02f3dba188ffa8f72780
7a802bda71da177ca7b55ed35fdae33bda98a81cee6731251ef092da2e91d75809b3db3c1a3432c2a430ab61ebc5f297774a880842cf0e43f93464821fc3b2b8dcdc2fc96371bf9df31b83b42e1b5b57014cc1d64f9d6b
NONCE=3e45a6de4c43d3f9c51fa2caf4ff9929caaa6038088bbedeb1388b334831afea80621f4aba0c7221b8bc02182b1ac05316d6a80f2005bceca3859a3145ec72fa4c40853c7d3dc4459f5079afe2c9b5a22de314f0bc128463ee0689cca536109c
POW_DIFFICULTY_LEVEL=252 TIMESTAMP=1519751894.29 MSG_HASH=0ce65c1874662ba9d64755a983db9056003a6065c80d068c4a1459df50489b6d
SELLER_SIGNATURE=68e5afedaa0f438586cc4995e8f5e5cfafd7dc51fa56e38f5f58a25537e49ae71e5246a2ef9df4a3a44bdb22cc89662acfd4f4e0ba0157b64cc32e1aebe8f075
```


How to Transfer the Ownership of a Coin:

- As was suggested by Satoshi Nakamoto, the ownership of a coin from a seller to a buyer can be established by the seller asking the buyer for his/her public key, incorporating that key as an additional field in the coin, and then hashing the thus augmented coin, and, finally, with the seller encrypting the augmented coin with his private key. In `CeroCoinClient`, after a node has mined a new coin, the node seeks a buyer for the coin

by reaching out to the rest of the nodes (actually a randomized list of the rest of the nodes). This is done by the method `find_buyer_for_new_coin_and_make_a_transaction()` shown below:

```
def find_buyer_for_new_coin_and_make_a_transaction(self):
    '''
    We now look for a remote client with whom the new coin can be traded in the form of a
    transaction. The remote client must first send over its public key in order to construct
    the transaction.
    '''
    mdebug = False
    while len(self.outgoing_client_sockets) == 0: time.sleep(2)
    while len(self.coins_currently_owned_digitally_signed) == 0: time.sleep(2)
    time.sleep(4)
    while True:
        while len(self.coins_currently_owned_digitally_signed) == 0: time.sleep(2)
        while self.blockmaker_flag or self.blockvalidator_flag: time.sleep(2)
        self.transactor_flag = True
        print("\n\nLOOKING FOR A CLIENT FOR MAKING A TRANSACTION\n\n")
        coin = self.coins_currently_owned_digitally_signed.pop()
        if coin is not None:
            print("\nNew outgoing coin:  %s" % coin)
        buyer_sock = None
        new_transaction = None
        random.shuffle(self.outgoing_client_sockets)
        sock = self.outgoing_client_sockets[0]
        if sys.version_info[0] == 3:
            sock.send(b"Send pub key for a new transaction\n")
        else:
            sock.send("Send pub key for a new transaction\n")
        try:
            while True:
                while self.blockmaker_flag or self.blockvalidator_flag: time.sleep(2)
                message_line_from_remote = ""
                while True:
                    byte_from_remote = sock.recv(1)
                    if sys.version_info[0] == 3:
                        byte_from_remote = byte_from_remote.decode()
                    if byte_from_remote == '\n' or byte_from_remote == '\r':
                        break
                    else:
                        message_line_from_remote += byte_from_remote
                if mdebug:
                    print("\n::::::::::::::message received from remote: %s\n" % message_line_from_remote)
                if message_line_from_remote == "Do you have a coin to sell?":
                    if sys.version_info[0] == 3:
                        sock.send( b"I do. If you want one, send public key.\n" )
                    else:
                        sock.send( "I do. If you want one, send public key.\n" )
                elif message_line_from_remote.startswith("BUYER_PUB_KEY="):
```

```
                    while self.blockmaker_flag or self.blockvalidator_flag: time.sleep(2)
                    buyer_pub_key = message_line_from_remote
                    print("\nbuyer pub key: %s" % buyer_pub_key)
                    new_transaction = self.prepare_new_transaction(coin, buyer_pub_key)
                    self.old_transaction = self.transaction
                    self.transaction = new_transaction
                    self.transactions_generated.append(new_transaction)
                    print("\n\nNumber of tranx in 'self.transactions_generated': %d\n" % len(self.transacti
                    print("\n\nsending to buyer: %s\n"  % new_transaction)
                    if sys.version_info[0] == 3:
                        tobesent = str(new_transaction) + "\n"
                        sock.send( tobesent.encode() )
                    else:
                        sock.send( str(new_transaction) + "\n" )
                    self.num_transactions_sent += 1
                    break
                else:
                    print("seller side: we should not be here")
            except:
               print("\n\n>>>Seller to buyer: Could not maintain socket link with remote for %s\n" % str(socket
            self.transactor_flag = False
            time.sleep(10)
```

Subsequently, the node that has a coin to sell invokes the following method for constructing a transaction:

```
    def prepare_new_transaction(self, coin, buyer_pub_key):
        '''
        Before the seller digitally signs the coin, it must include the buyer's public key.
        '''
        mdebug = False
        if mdebug:
            print("\n\nPrepareTranx::Buyer pub key:  %s\n" % buyer_pub_key)
        new_tranx = CeroCoinTransaction( transaction_id       = self.gen_rand_bits_with_set_bits(32),
                                         coin                 = str(coin),
                                         seller_id            = self.ID,
                                         buyer_pub_key        = buyer_pub_key,
                                         seller_pub_key       = ",".join(self.pub_key_string.split()),
                                         pow_difficulty       = self.pow_difficulty,
                                         timestamp            = str(time.time()),
                                       )
        digitally_signed_tranx = self.digitally_sign_transaction(new_tranx)
        return digitally_signed_tranx
```

- After a node has accumulated a certain designated number of outgoing transactions, it packages them into a block. In contrast

with a transaction that goes to one specific other node in the `CeroCoin` network, a block is broadcast to the entire network. Here are some very important considerations regarding blocks:

  - **A most important property of a block is the *blockchain length* that is associated it with it.**

  - **For the very first block that is broadcast to the network, the length of the blockchain is equal to the number of transactions in the block**.

  - **After a block is *accepted* by the network, the nodes in the network have no choice but to use that block as a genesis string for discovering the next coin.** Should a node decide to ignore the accepted block and continue using a previously accepted block in its coin mining algorithm (with, obviously, a shorter blockchain), any transactions and blocks created by the code would have associated with them shorter blockchain length. These would eventually be rejected by the network in favor of transactions and blocks with longer blockchains.

- The last of the block-related bullets listed above, uses the phrase "after a block is accepted by the network." **What is the meaning of a block being accepted by a node in the network?** What that means is that it should be possible for any node in the network — should the node choose to do so — to verify that a block is authentic.

- Regarding the authenticity of a block, note that every new block has embedded in it a signed hash of the previous block whose hash was used as a genesis string for the new coin mined that are in the transactions in the new block. Also note that every block has a unique identifier (BlockID) associated with it. That makes it a simple matter to verify that the new block is a valid construction from the previous block. **This process can be continued recursively backwards until you reach the genesis block — meaning the first block that was accepted by the network.** The current implementation of `CeroCoinClient` does not include the code needed for establishing the authenticity of a block. I plan to include it in a future version of the module.

- As mentioned previously, after a node has accumulated a designated number of transactions, it packages them into a block that is then broadcast to the entire network. As to how many transactions to pack into a block is a decision that each node must make for itself. The advantage for choosing a relatively large number for the number of transactions that go into a new block is that its acceptance by the network would bring to an end abruptly a large number of ongoing mining runs at the other nodes. However, there is also a risk associated with waiting too long for packaging the transactions into a new block — you may receive a new block from some other node that increases the length of the blockchain that you are currently using in the coin mining algorithm and you'll need to suddenly abandon your ongoing mining search and start a fresh one with a new genesis string.

- Shown below is the `CeroCoinClass` method that is in charge of packaging the transactions in a new block and broadcasting the block to the network:

```
    def construct_a_new_block_and_broadcast_the_block_to_cerocoin_network(self):
        '''
        We pack the newly generated transactions in a new block for broadcast to the network
        '''
        mdebug = False
        time.sleep(10)
        while len(self.transactions_generated) < self.num_transactions_in_block: time.sleep(2)
        while True:
            while len(self.transactions_generated) < self.num_transactions_in_block: time.sleep(2)
            self.blockmaker_flag = True
            print("\n\n\nPACKING THE ACCUMULATED TRANSACTIONS INTO A NEW BLOCK\n\n")
            current_block_hash = min_pow_difficulty = None
            if self.block is None:
                current_block_hash = self.gen_rand_bits_with_set_bits(256)
                min_pow_difficulty = self.pow_difficulty
                self.blockchain_length = len(self.transactions_generated)
            else:
                current_block_hash = self.get_hash_of_block(self.block)
                min_pow_difficulty = 0
                for tranx in self.transactions_generated:
                    tranx_pow = int(self.get_tranx_prop(self.block, 'POW_DIFFICULTY'))
                    if tranx_pow  > min_pow_difficulty:
                        min_pow_difficulty = tranx_pow
                self.blockchain_length += len(self.transactions_generated)
            new_block = CeroCoinBlock( block_id            =  self.gen_rand_bits_with_set_bits(32),
                                       block_creator       =  self.ID,
                                       transactions        =  str(self.transactions_generated),
                                       pow_difficulty      =  min_pow_difficulty,
                                       prev_block_hash     =  current_block_hash,
                                       blockchain_length   =  self.blockchain_length,
                                       timestamp           =  str(time.time()),
                                     )
            self.transactions_generated = []
            new_block_with_signature = self.digitally_sign_block( str(new_block) )
            print("\n\n\nWILL BROADCAST THIS SIGNED BLOCK: %s\n\n\n" % new_block_with_signature)
            self.block = new_block_with_signature
            for sock in self.outgoing_client_sockets:
                if sys.version_info[0] == 3:
                    sock.send("Sending new block\n".encode())
                else:
                    sock.send("Sending new block\n")
                try:
                    while True:
                        message_line_from_remote = ""
                        while True:
                            byte_from_remote = sock.recv(1)
```

```
                        if sys.version_info[0] == 3:
                            byte_from_remote = byte_from_remote.decode()
                        if byte_from_remote == '\n' or byte_from_remote == '\r':
                            break
                        else:
                            message_line_from_remote += byte_from_remote
                    if mdebug:
                        print("\n::::::::::BLK: message received from remote: %s\n" % message_line_from_rem
                    if message_line_from_remote == "OK to new block":
                        if sys.version_info[0] == 3:
                            tobesent = self.block + "\n"
                            sock.send( tobesent.encode() )
                        else:
                            sock.send( self.block + "\n" )
                        break
                    else:
                        print("sender side for block upload: we should not be here")
            except:
                print("Block upload: Could not maintain socket link with remote for %s\n" % str(sockx))
        self.blockmaker_flag = False
        time.sleep(10)

def get_hash_of_block(self, block):
    (transactions,prev_block_hash,timestamp) = self.get_block_prop(self.block, ('TRANSACTIONS',
                                                                                 'PREV_BLOCK_HASH',
                                                                                 'TIMESTAMP'))
    return message_hash(transactions + prev_block_hash + timestamp)
```

- The reader is asked to visit the documentation page at

  `https://engineering.purdue.edu/kak/distCC/CeroCoinClient-2.0.1.html`

  for further information regarding the `CeroCoinClient` module regarding how precisely the module is implemented. As mentioned in the documentation page, a working demonstration of the principles that underlie crypto currencies requires an implementation that must either be multithreaded or one that is based on multiprocessing. There are two important reasons for that: **(1)** The search for a new coin is a completely random process in which the time taken to discover the next coin cannot be predicted in advance. And **(2)** The ongoing search for a new coin must be abandoned immediately when a block of transactions is received

from the network if the received block has a blockchain that is longer than what is being used in the current search, or, for a given blockchain length, if the received block has a POW difficultly that is greater than what is being used in the current search.

- The reasons mentioned above imply that, at the least, you must execute the coin mining code and the code that is in charge of receiving and validating incoming blocks in two separate threads or processes. Since a client can construct a new block using a mixture of transactions, some of which are based on the coins discovered by the client while others are based on the coins acquired from other clients, you are going to have to also run the block construction code in a separate thread or a process of its own.

- I have chosen a multithreaded implementation for CeroCoinClient that launches the following six different threads when you fire up the code in the CeroCoinClient module: **(1)** A server thread that monitors the server port 6404 for requests for connection from other CeroCoinClient hosts in a network; **(2)** When the request for a connection is accepted by the server socket running in the server thread, it hands over the connection to a client connection thread that maintains that connection going forward; **(3)** A network scanner thread that establishes connections with the other CeroCoin hosts in the network; **(4)** A coin miner thread that is constantly searching for a new coin at a specified level of

PoW difficulty; **(5)** A transaction maker thread that springs into action when a client discovers a new coin, offering the coin to the list of CeroCoin clients discovered by the scanner thread after this list is randomized; and, finally, **(6)** A block maker thread that springs into action when it sees a certain minimum number of transactions for packaging into a block.

15.10: HASH FUNCTIONS FOR EFFICIENT STORAGE OF ASSOCIATIVE ARRAYS

- While our focus so far in this Lecture has been on hashing for message authentication, I'd be remiss if I did not touch even briefly on the other extremely important use of hashing in modern programming — efficient storage of associative arrays. In general, the hash functions used in message authentication are different from those used for efficient storage of information and it is educational to see the reasons for why that is the case. The goal of this section is to focus on this difference by presenting examples of hash functions for efficient storage. I'll start with the concept of an associative array because that is what is stored in the containers based on hash functions.

- An associative array, also known as a map, is a list of *<key,value>* pairs. You run into these sorts of arrays all the time when solving practical problems. For an illustrative example, you can think of a telephone directory as an associative array that consists of a list of *<string,number>* pairs.

- When working with associative arrays, the goal frequently is to

store them in such a way that the *value* associated with a *key* can be retrieved in constant time, meaning in time that is independent of the size of the associative array. [Just imagine the practical consequences when that is not the case. What if the search program being used by a telephone operator responding to your query for the phone number for an individual had to linearly scan through the entire directory to fetch that number? In a large metropolitan area with tens of millions of people, a linear scan (or even binary search) through alphabetized sub-lists would take far too long.]

- These days all high-level programming and scripting language provide such efficient storage structures. Examples include `dict` in Python, `hash` in Perl, `HashMap` in Java, `Map` in C++, etc. Storage structures, in general, are referred to as *containers* and these would be examples of containers that are based on hashing.

- The basic data abstraction used in efficient storage of associative arrays is that of a *bucket* and the number of buckets in a storage container is referred to as the container's *capacity*. For each <*key,value*> that needs to be stored in the container, we want to hash the key to a bucket address. You would then place the <*key,value*> in question in that bucket. To state it more precisely, you would place a pointer to that <*key,value*> in a linked list at that bucket address.

- The main challenge for a hash function that maps keys to bucket addresses is to ensure that all the keys are as uniformly distributed as possible over all the available bucket addresses. Ide-

ally, you would want each bucket to contain a single <*key,value*> pair. When that is the case, then, at search time, you would apply the same hash function to the key you are interested in and the resulting bucket address would take you directly to the value you are looking for.

- When the keys are themselves integers, it is relatively easy to come up with hash functions that can distribute the keys more or less uniformly over the bucket addresses. Using the arguments in Section 10.5 of Lecture 10, we could set the capacity of the container to a large prime number and calculate the bucket address for a given key as the remainder modulo the prime (after multiplying the key with a small integer constant). Since such remainders are likely to be distributed uniformly over the range $(0, capacity)$, we can certainly expect that the buckets would be populated uniformly — provided the keys themselves are distributed uniformly over whatever range they occupy. [One of the earliest suggested approaches for hashing the keys for efficient storage of <*key,value*> pairs when the keys are strings was to just add the decimal values (as given by ASCII coding) associated with characters, calculate this addition modulo a prime number, and use the remainder as the hash index. This approach to hashing was suggested by Arnold Dumey back in 1956 in his book "Computers and Automation." **By the way, the first person to have coined the term "hash" was the IBM mathematician Hans Luhn in 1953.**]

- Until recently, several programming languages used the FNV hash function for their hash based containers. Based on the idea

of prime numbers mentioned above, FNV is fast, in the sense that it requires only two operations, one XOR and one multiply, for each byte of a key. Here is a pseudocode description of the FNV hash function:

```
hash = offset_basis

for each octet_of_data to be hashed
    hash = hash xor octet_of_data
    hash = hash * FNV_Prime
return hash
```

where `offset_basis` and `FNV_Prime` are specially designated constants. For example, for 32-bit based calculations, the function uses $\mathit{offset\_basis} = 2,166,136,261 = 0x811C9DC5$ and $\mathit{FNV\_Prime} = 2{**}24{+}2{**}8{+}0x93 = 16,777,619 = 0x01000193$. FNV stands for the last names of Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo, the inventors of the hash function.

- More recently, though, several of the programming languages that previously used the FNV hash have switched over to SipHash created by Jean-Philippe Anumasson and Daniel Bernstein on account of its much superior collision resistance. As you will recall, in the context of hashing, collision refers to multiple keys hashing to the same bucket address.

- When a hash function calculates bucket addresses modulo a large prime, you can run into high collision rates if the keys are such that, when translated into integers, the bit patterns associated

with them occupy mostly the high-level bits. You see, the modulo operation, by its definition, discards a certain number of high-level bits from the keys. For illustration, consider calculating key values modulo 256 and assume that all the keys when translated into integers have values larger than 256. In this case, since the remainders would all be zero, you will have all the <*key,value*> pairs placed in the bucket with address 0. Although such an extreme non-uniformity in the distribution of the keys over the buckets does not happen when the capacity is a prime, you may nonetheless end with an unacceptable level of collisions in certain buckets if the the low-level bits of the keys are mostly zeros.

- It is educational to see how Java hashes keys to bucket addresses in order to get around the above mentioned problem of too many collisions in some of the buckets. Java has two levels of hashing: (1) It associates a 4-byte hashcode with every class type object. These include instances that you create in your own code from class definitions and also objects such as the class definitions themselves that come with the language or that you create. And (2) It carries out supplemental hashing of the object-specific hashcodes to distribute the keys more or less uniformly over all the buckets.

- In Java, the hashcode associated with a regular integer, as constructed from the class `Integer`, is the integer value itself. If the bucket addressing was based solely on these hashcode, you'd obviously run into the collision problem described above. The

hashcode associated with with a `Long` is the XOR of the upper 4 bytes with the lower 4 bytes of the 8-byte object. The hashcode associated associated with a string is given by

```
public int hashCode() {
    int h = hash;
    // In the next block, 'value' is an array of chars in the String object
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];   // val(i] is the ascii code for i-th char
        }
        hash = h;
    }
    return h;
}
```

This hashcode calculation for a string `s` of size `n` characters boils down to:

```
s[0]*(31**(n-1))  +  s[1]*(31**(n-2))  +  ...  +  s[n-1]
```

- That brings us to the second round of hashing — supplemental hashing — that Java uses to calculate the bucket addresses. The goal of this round is to disperse the keys over the entire capacity. Here is Java's function for supplemental hashing

```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

where `h` is the hashcode associated with the object. As mentioned earlier, the goal of supplemental hashing is to disperse the keys

— even the keys that reside mostly in the upper range of the hashcode values — over the full capacity of the container. The operator '>>>' is Java's bitwise non-circular right shift operator.

- I must also mention the critical role that is played by Java's auto-resizing feature of the hash-based containers. Java associates a *load-factor* with a container that, by default is 0.75, but can be set by the user to any fraction of unity. When the number of buckets occupied exceeds the load-factor fraction of the capacity, Java automatically doubles the capacity and recalculates the bucket addresses for the items currently in the container. The default for capacity is 16, but can be set the user to any desired value.

15.11: HOMEWORK PROBLEMS

1. 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?

2. 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?

3. 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?

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

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

6. 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.

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

8. **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.

9. **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.