Network Security

Professor Gaurav S. Kasbekar
Department of Electrical Engineering
Indian Institute of Technology, Bombay

Week - 03

Lecture - 19

Message Integrity, Cryptographic Hash Functions and Digital Signatures: Part 4

Hello, recall that in the previous few lectures, we discussed message integrity, cryptographic hash functions, and digital signatures. We now discuss an example of a cryptographic hash function, namely SHA-1 or Secure Hash Algorithm-1. We'll discuss how it is computed. We'll see the details of how the SHA-1 cryptographic hash function is computed. And then, we'll discuss an attack on cryptographic hash functions known as the length extension attack.

So, SHA-1 or Secure Hash Algorithm-1 operates as follows. It breaks up the input into blocks of 512 bits each and processes these blocks one by one. And the output is a 160-bit hash value. At a high level, SHA-1 operates by mangling bits in a sufficiently complicated way so that every output bit is affected by every input bit. This is because we have seen that for a given input, the output should appear to be a completely random function of the input.

So, the output should be chosen from among the 2^l possible outputs uniformly at random for any given input. So, there should be a complicated relation between the input and the output of a cryptographic hash function. So, hence, SHA-1 mangles bits in a complicated way. Some example hashes are as follows. The hash value of the input string, which is given by this, is shown here.

So, this is the hash value, SHA-1 of this input. Now, even if we make a small change in the input, then, with a high probability, many output bits change. For example, in this word, 'dog', if we just change one letter in it to 'cog', and we find out the hash of the modified sentence, then we get a hash with different values for 81 of the 160 bits. So, this shows the result. This is the same sentence as the first sentence except that one letter is changed, so this 'D' is changed to 'C'. Then the hash value is shown here.

So, in hex, it is this value. So, notice that this looks completely different from this. In particular, it can be checked that 81 of the 160 bits are flipped. Hence, this hash function behaves like a completely random mapping from the input to the output. For every input, it appears as though the output is selected uniformly at random from the 2^{160} possible outputs.

So, the operation of SHA-1 is as follows. First, it pads the input message by adding a 1 bit at the end, followed by a string of zeros. How many zeros are added? As many zeros are added as necessary to make the length a multiple of 512 bits. But at least 64 zeros are added.

So, for example, if the input message is 511 bits in length, then a 1 bit will be added at the end, making it 512 bits in length, and then zeros are added to make the message a multiple of 64. However, since it is already a multiple of 64 (512), but then the rule is that at least 64 zeros must be added. So, hence, 512 more zeros are added to make the message 1024 bits in length. So, this is the rule: first, we add a 1 bit, and then we add a string of zeros—at least 64 zeros—such that the length is a multiple of 512 bits. Next, we take a 64-bit number containing the message length before padding and then OR it into the low-order 64 bits. So, notice that the low-order 64 bits are always zeros because of this rule here.

So, OR the length of the 64-bit number into the low-order 64 bits of the message to get the original message along with the padding, and the last 64 bits now contain the length of the message. This is known as "Merkle-Damgard strengthening". So, we omit the details, but one security weakness is easy to see. If, instead of this particular padding, we were only to pad the input message with zeros to make the total number of bits a multiple of 512, then two input messages that differed only in one or more zeros at the end and were of the same length after padding would result in exactly the same hash value. So, that's one reason we use Merkle-Damgard strengthening.

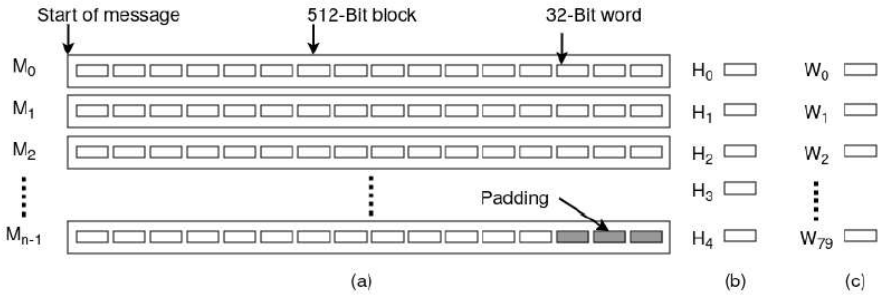
We ORed the length of the message, represented using 64 bits, into the low-order 64 bits. So, this is one reason, but we do not discuss the detailed reasons for the scheme, Merkle-Damgard's strengthening. So, we omit the details. These are published online, the reasons for using this particular strengthening. So, you can look it up if you're interested.

Fine, so during computation, SHA-1 maintains five 32-bit variables, H_0 to H_4, in which the hash value accumulates. This shows the input message. It is broken into chunks of 512 bits in length each. So, the chunks are M_0, M_1, M_2 up to M_{n-1}. And SHA-1 maintains five 32-bit variables.

These variables are H_0 to H_4, which are shown here. These are registers. And the hash value accumulates in these registers. These are initialized to some constants that are specified in the SHA-1 standard. We do not specify these constants.

You can look them up in the standard description. Five scratch variables or rough variables, A to E, are initialized to H_0 to H_4 respectively. Each of the blocks, M_0 to M_{n-1}, is now processed in turn. We take the i^{th} chunk, that is, M_i, and then process it in the way that we'll describe, and then we take the next chunk, M_{i+1}, and so on and so forth. So, we take each of these blocks and process it in turn.

- Five scratch variables, A to E, are initialized to H_0 to H_4, respectively
- Each of the blocks M_0 to M_{n-1} now processed in turn



So, for the current block M_i, the 16 32-bit words are first copied into the start of an 80-word array, W. This array, shown here, has 80 words, each of 32 bits. So, W_0, W_1, up to W_{79}, that's this array. So, the current chunk M_i has 16 words, each of 32 bits. These 16 words are shown here by these rectangles. So, these 16 words are copied into the start of this array.

So, they will be copied into W_0 to W_{31}. Then, the other 64 words in W are filled in using the following formula. $W_i = S^1(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})$. So, for $16 \leq i \leq 79$, W_i is calculated using this formula, where $S^b(W)$ represents the left circular rotation of the 32-bit word W by b bits. So, this is a 1-bit left circular rotation.

- $W_i = S^1(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})$, $16 \leq i \leq 79$
- where $S^b(W)$ represents the left circular rotation of the 32-bit word W by b bits

And this XOR represents a bitwise XOR. Next, the scratch variables A to E are assigned values using the following procedure. 'for(i = 0; i < 80; i++)', that is, we have a 'for' loop. First, we initialize a temporary variable to this. And then, we assign the variables A to E using these formulas.

$$\text{for } (i = 0; i < 80; i++) \{$$
$$\quad \text{temp} = S^5(A) + f_i(B,C,D) + E + W_i + K_i;$$
$$\quad E = D; D = C; C = S^{30}(B); B = A; A = \text{temp};$$
$$\}$$

So, we can see that we are performing some complicated-looking operations on the input message. And these constants K_i, which appear here, are defined in the standard. Again, we do not provide their values here for brevity. The functions f_i are defined as follows. So, we have the functions f_i for $0 \leq i \leq 79$.

- The functions f_i are defined as:
  - f_i (B,C,D) = (B AND C) OR (NOT B AND D) $(0 \leq i \leq 19)$
  - f_i(B,C,D) = B XOR C XOR D $(20 \leq i \leq 39)$
  - f_i(B,C,D) = (B AND C) OR (B AND D) OR (C AND D) $(40 \leq i \leq 59)$
  - f_i(B,C,D) = B XOR C XOR D $(60 \leq i \leq 79)$

Their values are specified as these. f_i is given by this value for $0 \leq i \leq 19$, and it is given by this value for $20 \leq i \leq 39$, and this value for $40 \leq i \leq 59$, and this value for $60 \leq i \leq 79$. So, again, these look like very complicated processing of the inputs to get the intermediate values. When all 80 iterations of the above loop are completed, A to E are added to the registers H_0 to H_4. So, this completes the processing of the current 512-bit block.

So, we have now discussed how a particular chunk of 512 bits is processed. Then, the next chunk is taken, and the same processing is done on it. So, each time a chunk M_i is processed, these registers H_0 to H_4 keep getting updated. So, they are initialized to some constants that are specified in the standard. Then, the first chunk M_0 is processed and then the values of H_0 to H_4 change to some other values.

Then, the next chunk M_1 is taken, and these registers further change, and so on and so forth. So, when the last chunk M_{n-1} is processed, after this, the registers H_0 to H_4 have certain values, and they are output as the final output of the hash function. So, to process the next

chunk, the W array is reinitialized from the new block, and H is left as it was. When this 512-bit block is finished, the next one is started, and so on, until all the 512-bit blocks have been processed. When the last block has been finished, the five 32-bit words in the H array are output as the 160-bit cryptographic hash.

So, after the last block, that is M_{n-1} in this figure, after this has been processed, whatever value remains in these registers H_0 to H_4, that is the cryptographic hash function that consists of 160 bits. So, recall that a cryptographic hash function with l bits as the output, for example, l may be 160, as in the case of SHA-1. Such a cryptographic hash function must appear to be such that for any given input, the output is selected uniformly at random from the 2^l possible outputs. For achieving this, a cryptographic hash function is a series of steps, such as the ones that we saw in the case of SHA-1, and each step mangles the message more and more. So, we saw examples of such steps when we discussed SHA-1 in the previous few slides.

So, now the crucial question is: why is this particular mangling of the message done? So, we saw different operations that were performed. For example, here, why are these functions fi chosen in these particular ways? There are many other logical operations we could have performed to get some functions. Why are these particular functions used?

To ask another question, why are these particular operations performed to get 'temp' and then why are E to A set in this particular way? So, this is not a unique mapping. There are many cryptographic hash functions which could have worked. So, going back to the question that we are discussing. So, the reason that these particular functions are used and these particular operations are used is as follows.

Recall that the objectives when designing a cryptographic hash function are as follows. The output should appear to be a completely random function of the input. And the function must satisfy the conditions in the definition of a cryptographic hash function. For example, it should be computationally infeasible to find an input corresponding to a given hash value. And the function should be easy to compute.

So, it should not be computationally expensive. And similar to a symmetric key encryption algorithm, most cryptographic hash functions are computed in rounds. So, a plausible way to design a cryptographic hash function is as follows. We combine various mangling operations, such as the ones that we saw in the previous few slides, to get a potential cryptographic hash function. And then, we check whether the output passes some randomness tests.

For example, for a random input, roughly half the output bits are zeros and the others are ones. And we check whether the output passes some security tests. For example, is it computationally feasible to find an input corresponding to a given hash value? If this function does not pass these randomness tests or security tests, then we add some more mangling operations, change some operations, or try some other potential cryptographic hash function. And we keep on repeating this process.

So, several designers tried out various mangling operations, and they ran the security tests and randomness tests on different potential cryptographic hash functions. And the design of SHA-1 that we discussed in the previous few slides is one function which was found to work. That is, it satisfies randomness tests and security tests. So, that is one choice which is found to work. And so far, no one has found a way to break the security, and it has been found to satisfy randomness tests.

So, it is currently being used in practice. Potentially, in principle, one could use some other operations to get another cryptographic hash function, and indeed, there are many cryptographic hash functions which are used in practice. We now discuss an attack known as the length extension attack, which can be performed when a cryptographic hash function, such as SHA-1, is used. So, recall the protocol that we discussed earlier for achieving message integrity. Assume that Alice and Bob have a shared secret bit string s. Alice performs the following actions.

She concatenates m and s to get (m,s) and computes its hash, H(m,s), and then sends m along with H(m,s) to Bob. Bob computes H(m,s) using m and s and checks whether it equals H(m,s) sent by Alice. So, recall that this H is the message authentication code. This protocol successfully achieves message integrity. Now, let's modify this protocol slightly.

- Alice performs the following actions:
  - concatenates m and s to get (m, s); computes $H(m, s)$
  - sends $(m, H(m, s))$ to Bob
- Bob computes $H(m, s)$ using m and s; checks whether it equals $H(m, s)$ sent by Alice

Instead of sending (m, H(m,s)), suppose Alice sent (m, H(s,m)). So m and s are, the order is reversed, so we find out H(s,m) instead of finding H(m,s). And Alice sends this information to Bob. And suppose that the hash function H(.) used here is SHA-1 or a similar cryptographic hash function. Then the claim is that this modified protocol would be

vulnerable to an attack known as the 'Length Extension Attack'. So, we'll see that this original protocol is not vulnerable to the length extension attack, but if we use this modified protocol where Alice sends (m, H(s,m)) to Bob, then this is vulnerable to the length extension attack when the hash function used is SHA-1 or a similar hash function. So, we now describe the length extension attack.

So, by studying the operation of SHA-1 on the previous few slides, we infer the following. To compute the hash value of a padded message up to chunk n, we only need the hash value up to chunk n-1, and the value of chunk n. So, recall that the message was processed in chunks M_0, M_1, M_2, and so on. So, after completing the processing up to chunk M_i, we then take the resultant hash value and then process the chunk M_{i+1}, and then the resultant hash value plus the chunk M_{i+1} are enough to compute the hash value up to chunk M_{i+1}. So, hence, we have this property that to compute the hash value of a padded message up to chunk n, we only need the hash value up to chunk n-1 and the value of chunk n. So, this is the basis behind the length extension attack. Suppose Alice sends (m, H(s,m)) to Bob.

- An intruder on the path between Alice and Bob modifies the message to $(m, z, e, H(s, m, z, e))$, where
  - z is the pad that was added by Alice before computing hash value of (s, m)
  - e are the spurious bits that the attacker wants to append to message

An intruder on the path between Alice and Bob modifies the message to this message. (m, z, e, H(s, m, z, e)), where z is the pad that was added by Alice before computing the hash value of (s, m). And e are the spurious bits that the attacker wants to append to the message. So, the attacker wants to extend the length of the message by adding some spurious bits e at the end of the message. So, the original message is m, but the intruder wants to add some messages, some spurious bits e to the message. But then, the intruder cannot directly append e to m because then the intruder would not be able to compute the hash value of H(s, m, e).

So, the intruder has to first calculate the padding z that was used by the sender Alice, and then the intruder adds the padding z and then appends the spurious bits e to it, and then computes H(s, m, z, e). So, the crux of this is that to compute H(s, m, z, e), the intruder just needs to know H(s, m) along with the spurious bits e and the padding z. So, this can be computed using just this information. So, you can verify easily that the spurious bits, which the intruder wants to add, are clearly known. So once these spurious bits are known and the padding is not difficult to compute, once these are known, using this property, it's easy

for an intruder to calculate this modified message. The hash value verification is successful at the receiver Bob, and he accepts the modified message. So, this modified message along with its message authentication code can be found by the intruder if he/she knows the message m, which is sent in the clear.

Then the intruder needs to know H(s, m), which is also sent by Alice and the length of the secret string s. So, this length of the secret string s is required to calculate the padding. That is because the length of the message that is hashed is ORed into the low-order 64 bits of the message, as we saw during our discussion of SHA-1. So, the intruder needs to know the length of the secret string s in order to calculate the pad that needs to be added to the message, that is, the pad z. If the length of the secret string s is unknown to the intruder, then it can be found by trying out different lengths until Bob accepts the message, assuming that Bob provides an acknowledgment when he receives a valid message.

So, the intruder tries out different lengths of the secret string s until Bob sends an acknowledgment, in which case, the intruder has found the correct length of the secret string s. So, this is the length extension attack. This will become clearer when we discuss an example. Here's an example. A server for delivering waffles of a specified type to a specific user at a location accepts requests of the following format. The original data is this.

- *Original Data*: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo
- *Original MAC*: 6d5f807e23db210bc254a28be2d6759a0f5f5d99

Count=10, latitude equals this. User_id is 1 and longitude equals this. And waffle is eggo. And the original MAC is this. This is the message authentication code.

So, for simplicity, we have not used any secret string here. Now, the server checks the MAC and, if correct, performs the request given in this message, that is, to deliver 10 waffles of type eggo to the given location for user 1. The location is specified by these latitude and longitude values. Now, assume that if there are multiple waffle equals something fields, then the server uses the last such field and ignores the others. So, there is only one field, waffle equals eggo, but after a length extension attack, there will be multiple fields, and then the server will use the last such field and ignore the others.

It is possible for an attacker to modify this request and change the waffle type from eggo to liege. So, the desired new data is this. The new data consists of the original message along with &waffle equals liege added to the end. So, these are the spurious bits added by the intruder. The attacker modifies this message to the following.

- *Desired New Data*: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo**&waffle=liege**

The new data is the original message plus the padding that needs to be added plus the spurious bits. And the MAC is this. The intruder is able to calculate the MAC because the original MAC is known, and then the intruder has to use the above procedure in the previous slide to calculate the new MAC. So, the intruder has enough information to calculate the new MAC. So, this is an example.

So, notice that the intruder had to first add the padding and then add the spurious bits. So, the intruder is able to add these spurious bits at the end of the message. And the result of this message is that the wrong type of waffles is sent to the customer. That is, instead of waffle = eggo, waffle = liege is sent to the customer. So, the intruder is able to successfully modify the message and change its meaning.

So, this is an example of the length extension attack. We now discuss different defenses against the length extension attack. If Alice sends (m, H(m,s)) to Bob instead of (m, H(s, m)), then the protocol is not vulnerable to the length extension attack. That's because if H(m, s) is sent, then the intruder is not able to add spurious bits at the end of the message m and then still compute the MAC because there is a secret string s which needs to be added at the end of the message, which the intruder does not know. So, this is not vulnerable to the length extension attack.

Here is an alternative attempt to defend against the length extension attack. If we want to use H(s, m), assume for concreteness that SHA-1 is used to compute the hash. Alice wants to send the message m; she computes H(s, m), but now instead of sending the entire H(s, m), Alice sends (m, H_l(s, m)), where H_l(s, m) are the lower-order 80 bits of H(s, m). So, Alice sends only half of the bits of H(s, m), the lower-order 80 bits, and omits the upper-order 80 bits. So, this successfully defends against the length extension attack because the attacker does not have enough information to compute the hash of the modified message since the intruder does not know the upper-order 80 bits of H(s, m). So, by omitting the upper-order 80 bits of the message, one can defend against the length extension attack.

But the receiver of the message, that is, the legitimate receiver of the message, is able to verify the message authentication code because they know the secret s; they can concatenate s and m and find out its hash and then check whether the lower-order 80 bits match this value, H_l(s, m), sent by the sender. So, the receiver is able to verify the MAC,

but this scheme defends against a length extension attack. Now, SHA-1 and SHA-2 are cryptographic hash functions, which are vulnerable to the length extension attack. But a newer scheme, SHA-3, is not vulnerable to the length extension attack. The reason is that SHA-1 and SHA-2 process the input by splitting it into blocks or chunks and producing for each block an internal state, which is the same length as the hash function output.

That is, 160 bits in the case of SHA-1, for example. But in contrast, in SHA-3, the internal state is much larger than the hash function output. So, from the hash function output, the intruder is not able to recover the internal state. And hence, there is not enough information to extend the length of the message. So, hence, this protocol in which Alice sends (m, H(s, m)) to Bob, where H(.) is SHA-3 instead of SHA-1.

This protocol achieves message integrity. It is not vulnerable to the length extension attack. So, in summary, SHA-3 uses an internal state, which is much larger than the hash function output. For this reason, it is not vulnerable to the length extension attack. This concludes our discussion of message integrity, cryptographic hash functions, and digital signatures.

Thank you.