EE 418 Network Security and Cryptography
Lecture #14

November 17, 2016

Iterated Hash Functions. Message Authentication Codes (MACs).
Lecture notes prepared by Professor Radha Poovendran.

Tamara Bonaci
Department of Electrical Engineering
University of Washington, Seattle

Outline:

1 Review: Iterated Hash Functions

Last time, we started from the observation that in practice, communicating users Alice and Bob generate messages of varying length, and that, in turns, creates the need for hash functions that take inputs of arbitrary length.

A widely-used approach for hashing an arbitrary-length message is through *iterated hash functions*. The idea of the iterated hash is to start with a cryptographic hash function that is believed to be secure for fixed-length input (i.e., satisfies preimage, second preimage, and collision resistance), and construct an extended hash function that takes arbitrary-length input while preserving the security properties. For communication efficiency, the length of the hash output should remain fixed. One well-known approach, that provides both efficiency and security it the *Merkle-Damgård iterated hash function.*

1.1 Merkle-Damgård Iterated Hash Function

Merkle-Damgård takes as input a message m of length k, a collision-resistant hash function $h : \{0,1\}^{n+t+1} \rightarrow \{0,1\}^n$, and an optional post-processing function $f : \{0,1\}^n \rightarrow \{0,1\}^l$ that is publicly known. It consists of **Pre-processing**, **Processing**, and **Post-processing** phases, and these three steps of Merkle-Damgård are described as follows.

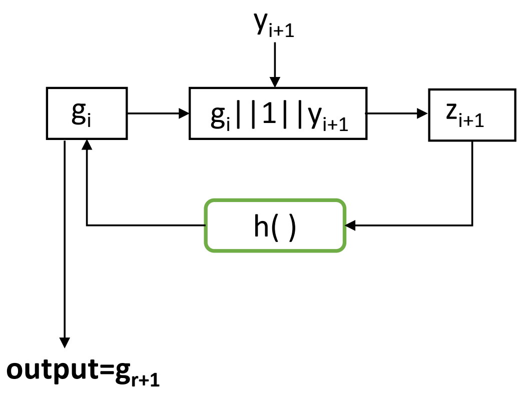

Fig. 1. Merkle-Damgård iterated hash function.

Pre-Processing In this stage, the message m is divided into blocks of length t, as follows:

1. Divide m into blocks $m = m_1 || \cdots || m_r$, where $|m_i| = t$ for $i = 1, \dots, (r-1)$ and $|m_r| \leq t$.
2. Compute $d = rt - k$, which is equal to the number of zeros that must be added to m_r so that $|m_r| = t$.
3. Define $y_1, \dots, y_r$ as $y_i = m_i$, $i = 1, \dots, r$, and $y_r = m_r || 0^d$.
4. Define y_{r+1} to be the binary representation of d.
5. Output $(y_1, \dots, y_{r+1})$.

Processing The processing stage consists of r steps, where the output of each step is a digest g_i, defined by:

$$g_1 = h(z_1), \qquad z_1 = 0^{n+1} || y_1 \tag{1}$$

$$g_i = h(z_i), \qquad z_i = g_{i-1} || 1 || y_i \qquad \text{for } i > 1 \tag{2}$$

Note that in (2), the input z_1 to the hash function has length $(n+1+t)$, and hence is a valid input to the hash function h. Furthermore, $(z_{i-1}||1||m_i)$ has length $(n+1+t)$, and hence is also a valid input. The bit '1' is inserted between z_{i-1} and m_i at each iteration in order to ensure collision resistance. The output of the processing phase is $g(z_{r+1})$.

Post-Processing The post-processing phase is optional. In this phase, suppose that $f : \{0,1\}^n \to \{0,1\}^l$ is a publicly known function. The hash value $\overline{h}(m)$ is given by $\overline{h}(m) = f(z_{r+1})$.

1.2 Security Analysis of Merkle-Damgård

Merkle-Damgård is known to be secure if the hash function $h : \{0,1\}^{n+t+1} \to \{0,1\}^n$ is collision-resistant, as described by the following theorem.

Theorem 1. *Suppose that $h : \{0,1\}^{n+t+1} \to \{0,1\}^n$ is a collision resistant hash function. Then the function $\overline{h}$ constructed using the Merkle-Damgård algorithm is collision-resistant.*

Proof. Proof of the collision resistant property of Merkle-Damgård function goes as follows: It assumes that the original **compress** function is collision resistant. It then shows that **if there is a collision** in Merkle-Damgård function, then the **original h have to have collision.**

Suppose that we can find $x \neq x'$ such that $\overline{h}(x) = \overline{h}(x')$. We will show that this leads to a polynomial-time algorithm for finding a collision for h. Let

$$y(x) = y_1 || y_2 || \cdots || y_{k+1}$$
$$y(x') = y'_1 || y'_2 || \cdots || y'_{l+1},$$

where $k+1$ and $l+1$ are the respective block lengths of x and x' which are padded by d and d' 0's, respectively. Denote the g-values computed in the algorithm for strings x and x^{prim} by $g_1, \ldots, g_{k+1}$ and $g'_1, \ldots, g'_{l+1}$, respectively. We identify two cases, depending on whether $|x| \equiv |x'| \bmod (t-1)$.

Case 1: In case 1, $|x|$ is not congruent to $|x'| \bmod (t-1)$. Here $d \neq d'$ and $y_{k+1} \neq y'_{l+1}$. We have

$$\begin{aligned} h(g_k || \mathbf{1} || y_{k+1}) &= g_{k+1} = \overline{h}(x) \\ &= \overline{h}(x') = g'_{l+1} \\ &= h(g'_l || \mathbf{1} || y'_{l+1}) \end{aligned}$$

which is a collision for h because $y_{k+1} \neq y'_{l+1}$.
Case 2: In case 2, the colliding strings have equal length. i.e. $|x| = |x'|$. Here, $k = l$ and $y_{k+1} = y'_{k+1}$. As in case 1,

$$\begin{aligned} h(g_k || \mathbf{1} || y_{k+1}) &= g_{k+1} = \overline{h}(x) \\ &= \overline{h}(x') = g'_{k+1} \\ &= h(g'_k || \mathbf{1} || y'_{k+1}). \end{aligned}$$

If $g_k \neq g'_k$, then we find a collision for h, so we assume that $g_k = g'_k$. Then we have

$$h(g_{k-1} || \mathbf{1} || y_k) = g_k = g'_k = h(g'_{k-1} || \mathbf{1} || y'_k).$$

Assuming we do not find a collision, we continue working backwards until we obtain $g_1 = {g_1}'$.

$$g_1 = h(0^{m+1} || y_1) = g'_1 = h(0^{m+1} || y'_1).$$

If $y_1 \neq y'_1$, then we find a collision for h. But h was assumed to be **collision resistant.** So we must have $y_1 = y'_1$. Hence, if **h is collision resistant,** we find that $y_i = y'_i$ for $1 \leq i \leq k+1$, and so the binary strings are equal (i.e. $x = x'$). But we started with the assumption that that $x \neq x'$. So we have a contradiction here. We conclude that if there is a collision in the iterated hash then the original h is not collision resistant.

Case 3: In case 3, $|x| \neq |x'|$ and $|x| \equiv |x'| \bmod (t-1)$. Without loss of generality, assume $|x'| > |x|$, so $l > k$. If there no collision for h, we eventually have

$$h(0^{m+1} || y_1) = g_1 = g'_{l-k+1} = h(g'_{l-k} || \mathbf{1} || y'_{l-k+1}).$$

However, the $(m+1)$-st bit of $0^{m+1} || y_1$ is a 0 and the $(m+1)$-st bit of $g'_{l-k} || \mathbf{1} || y'_{l-k+1}$ is a 1, implying that we have found a collision for h. This, however, contradicts the assumption that h is collision-resistant.

Hence if h is collision resistant, using this method we can generate hashes for arbitrary length messages without worrying about generating new collisions due to iteration.

2 Message Authentication Codes (MACs)

A message authentication code (MAC) is a code that is *appended to a message* in order to provide message integrity[1]. In MACs, communicating parties share a *secret key* that is used to generate the code, and if a MAC is well-designed, then only a user with the shared key can compute a valid MAC for a given message. Two parties Alice and Bob who share a key K can use a MAC for message integrity as follows (illustrated in Figure 2):

1. Alice computes a MAC for message m as $y = MAC(K, m)$, and sends pair (m, y) through the channel to Bob.
2. Bob receives (m, y). Using the secret key, Bob computes $MAC(K, m)$ and checks $y \overset{?}{=} MAC(K, m)$.
3. If $y = MAC(K, m)$, then Bob **accepts** the message. Otherwise, Bob **rejects** the message, since the message and MAC are inconsistent.

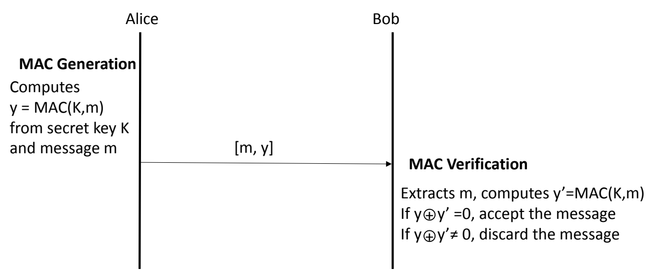

Fig. 2. Block diagram for message integrity using a message authentication code (MAC).

Note: Message authentication codes introduce a secret key into the hash function computation in order to provide message integrity. A MAC is secure if a third-party Eve cannot generate m' and y' satisfying $y' = MAC(K, m')$ without knowledge of the secret key K. If Eve can generate such a pair (m', y'), then this pair is a *forgery*.

There are two basic methods for constructing MACs. In the first method, the MAC is constructed by incorporating a key K into the hash function computation. In the second method, the MAC is constructed by using block encryption techniques. We now describe a standard MAC construction of each type.

2.1 MAC from a Hash Function: HMAC

The most popular method for constructing a MAC from a hash function is HMAC, which was developed by Mihir Bellare, Ran Canetti, and Hugo Krawczyk in 1996. HMAC has been implemented in standard protocols such as Transport Layer Security (TLS) and IPsec, and has been adopted by NIST as FIPS 198. The **basic idea** of HMAC is to compute

$$HMAC(K, m) = h(f(K, m)),$$

where h is a hash function and f is a publicly-known function. This MAC construction provides the following security properties:

- If the hash function is second preimage- and collision-resistant, and f is well-chosen, it will be difficult for an attacker who does not know K to compute (m, y) with $y = h(f(K, m))$.
- If the hash function is preimage-resistant, and f is well-chosen, it will be difficult for an attacker to compute K given $h(f(K, m))$, as this would require inverting the hash function.

A pseudocode description of HMAC is given as Algorithm 1, and a schematic illustration is given as Figure 3.

HMAC takes as input a key K, a hash function $h : \{0,1\}^n \to \{0,1\}^l$, and a message $m = m_1 || \cdots || m_r$, where $|m_i| = n$ for all i (m_r can be padded with zeros to length n), and it generally consists of two steps, **Inner Padding** and **Outer Padding** steps, are described as follows.

1. **Inner Padding:** The key K is padded with zeros on the left so that $|K| = n$. A bit string *ipad* is generated, consisting of repetitions of the eight-bit pattern 00110110:

$$ipad = \underbrace{00110110 \quad 00110110 \quad \cdots \quad 00110110}_{n \text{ bits total}}.$$

   The number of repetitions is chosen such that $|ipad| = n$. Compute $S_i = K \oplus ipad$. The output of this phase is the hash digest $x = h(S_i || m_1 || \cdots || m_r)$.

[1] Unlike a regular hash function, where the hash output needs to be exchanged via a secure side channel.

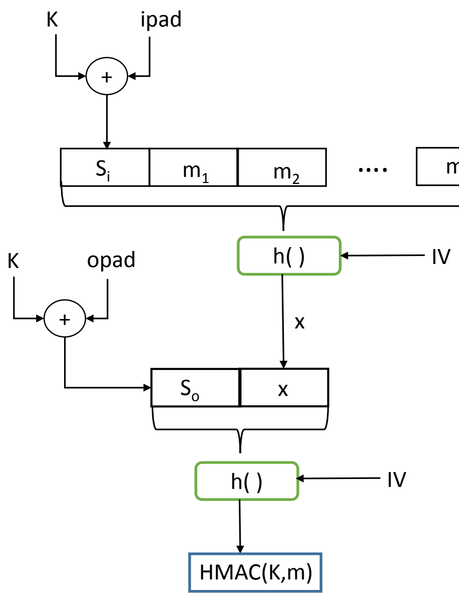

Fig. 3. HMAC algorithm for constructing a MAC from a hash function.

Algorithm 1 Constructing a MAC from a hash function.

```
 1: procedure HMAC($m$, $h$, $K$, $IV$)
 2:     Input: Message $m$ of length $|m| = k$, $m = m_1||\cdots||m_r$ where $|m_i| = n$.
 3:       Key $K$
 4:       Collision-resistant hash function $h : \{0,1\}^n \to \{0,1\}^n$
 5:       Vector $IV$ used to compute hash function.
 6:     Output: MAC value $HMAC(K, m)$
 7:     Pad $K$ with zeros on the left so that $|K| = n$
 8:     $t \leftarrow n/8$
 9:     $ipad \leftarrow t$ copies of 00110110
10:     $S_i \leftarrow K \oplus ipad$
11:     $x \leftarrow h(S_i||m_1||\cdots||m_r)$
12:     $opad \leftarrow t$ copies of 01011100
13:     $S_o \leftarrow K \oplus opad$
14:     $HMAC(K, m) \leftarrow h(S_o||x)$
15:     return $HMAC(K, m)$
16: end procedure
```

2. **Outer Padding:** A bit string *opad* is generated, consisting of repetitions of the eight-bit pattern 01011100:

$$opad = \underbrace{01011100 \quad 01011100 \quad \cdots \quad 01011100}_{n \text{ bits total}}.$$

   The number of repetitions is chosen such that $|opad| = n$. Compute $S_o = K \oplus opad$. The output of this phase is

$$HMAC(K, m) = h(S_o||x).$$

It can be shown that if the hash function h is preimage-, second preimage-, and collision-resistant, then it is computationally hard for an adversary to create a forgery $y' = HMAC(K, m')$ for message m without knowledge of key K. This proof of security is one of the main benefits of HMAC.

2.2 MAC from Block Cipher: CBC-MAC

Another approach for constructing MACs is using block ciphers. An advantage of using block ciphers to compute a MAC is that the same hardware or software implementation can be used for both encryption and authentication (but the best practice is to use different keys for encryption and message authentication). Block ciphers can also be used to develop *authenticated encryption* mechanisms, which provide both message confidentiality and integrity.

The most common MAC construction using block ciphers is CBC-MAC, which is recommended by NIST 800-38B and NIST 800-38C. CBC-MAC computation is illustrated in Figure 4 and as Algorithm 2. CBC-MAC uses a symmetric key encryption function E_K as a building block, with $E_K : \{0,1\}^n \rightarrow \{0,1\}^n$ (examples are DES with $n = 64$ or AES with $n = 128$). The steps in CBC-MAC computation are as follows:

1. The message m is divided into r blocks, each of length n, denoted $m_1, \ldots, m_r$. The last block is padded with zeros to ensure that the length is n.
2. The first iteration is computed as $y_1 = E_K(m_1 \oplus IV)$, where IV represents an initialization vector.
3. For each $i = 2, \ldots, r$, compute $y_i = E_K(m_i \oplus y_{i-1})$.
4. The MAC value is given by CBC-MAC$(K, m) = y_r$.

Note that, unlike the CBC encryption mode, only the last block y_r is transmitted. As in CBC encryption mode, either a fixed value of IV can be used, or a random value of IV can be used and sent along with the message.

Algorithm 2 MAC construction using CBC encryption mode.

```
procedure CBC-MAC($m$, $h$, $K$, $IV$)
    Input: Message $m$ of length $|m| = k$, $m = m_1||\cdots||m_r$ where $|m_i| = n$.
        Key $K$
        Symmetric-key encryption function $E_K : \{0,1\}^n \rightarrow \{0,1\}^n$.
        Vector $IV$ used to compute hash function.
    Output: MAC value CBC-MAC$(K, m)$
    $y_1 \leftarrow E_K(m_1 \oplus IV)$
    for $i = 1, \ldots, (r-1)$ do
        $y_{i+1} \leftarrow E_K(m_{i+1} \oplus y_i)$
    end for
    CBC-MAC$(K, m) \leftarrow y_r$
    return $y_r$
end procedure
```

CBC - MAC Example: Let's assume that Alice wants to talk to Bob, and they are very concerned about the integrity of their message. They have agreed to an initialization vector $IV = y_0 = 0000$, secret key $K = 1101$, an encryption method $e_K(x) = x \oplus K$, and a block of size four, $t = 4$.
If the message x that Alice wants to send is given as $x = 10011101$ 1011 0110 1111, then:

(a) Alice first splits message into blocks of size $t = 4:\ \ x = 1001\ 1101\ 1011\ 0110\ 1111$.
(b) She then sends to Bob: $x||h_K(x) = 1001$ 1101 1011 0110 1111 **1011**.

Questions:

(a) How does Bob know the message length?
(b) Where should Alice put that information?

Answer: Alice and Bob need to agree about the maximum size of a message or block size.

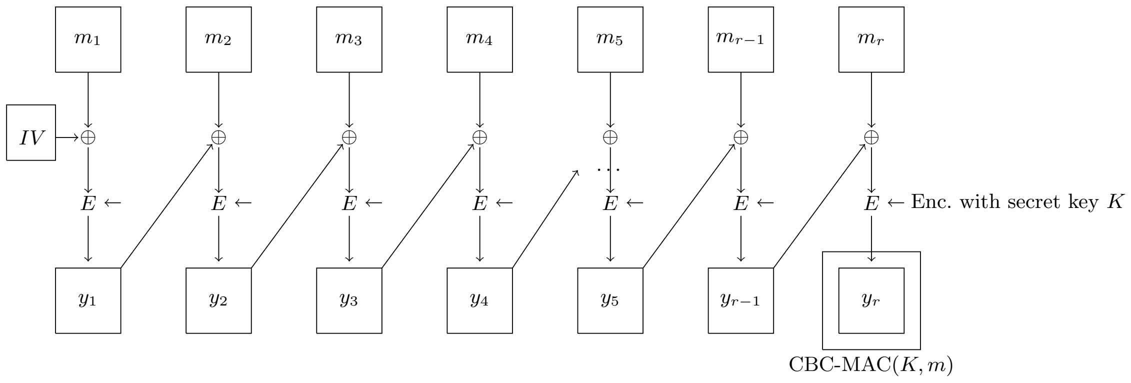

Fig. 4. CBC-MAC computation.

3 Applications of Hash Functions

We have already seen one application of keyed hashes for the generation of MACs, but there are a few other applications, and we explore them next.

3.1 Source Authentication with a Hash Chain

Assume that a company wants to to store a file with all the passwords of its clients. The simplest way of doing so is by listing all the passwords in one file. If the file gets stolen, all the passwords will be compromised. An alternative way to that would be to store the hashed value of the password instead of the actual password value. Then, when a user wants to log in into the system, it inputs his/her password, and the hash value of that password is checked against the stored value. If the password file gets stolen, an attacker would not be able to recover the passwords from the file, unless it solved the preimage or collision problem.
Question: Do you see any problem with this type of authentication?

Sources for Today's Lecture:

1. Douglas R. Stinson, *Cryptography, Theory and Practice, 3rd edition.* CRC Press, 2005, p. 119–155 and 393–453.
2. Wade Trappe and Lawrence C. Washington *Introduction to Cryptography with Coding Theory.* Prentice Hall, 2002, p. 182–186 and 236–246.
3. Charlie Kaufman, Radia Perlman, and Mike Speciner *Network Security: Private Communication in Public World, 2nd Edition.* Prentice Hall, 2002, p. 117–143 and 147–165 and 307–365 and 371–401.