Hash functions and Digital Signatures.

Foros Valentin

December 4, 2022

1 Theory

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string. A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. To prevent the conversion of hash back into the original key, a good hash always uses a one-way hashing algorithm. A good hash algorithm should be complex enough such that it does not produce the same hash value from two different inputs. If it does, this is known as a hash collision. A hash algorithm can only be considered good and acceptable if it can offer a very low chance of collision.

2 SHA256

SHA 256 is a part of the SHA 2 family of algorithms, where SHA stands for Secure Hash Algorithm. Published in 2001, it was a joint effort between the NSA and NIST to introduce a successor to the SHA 1 family, which was slowly losing strength against brute force attacks. The significance of the 256 in the name stands for the final hash digest value, i.e. irrespective of the size of plaintext/cleartext, the hash value will always be 256 bits.

Some of the standout features of the SHA algorithm are as follows:

1. Message Length: The length of the cleartext should be less than 264 bits. The size needs to be in the comparison area to keep the digest as random as possible.
2. Digest Length: The length of the hash digest should be 256 bits in SHA 256 algorithm, 512 bits in SHA-512, and so on. Bigger digests usually suggest significantly more calculations at the cost of speed and space.
3. Irreversible: By design, all hash functions such as the SHA 256 are irreversible. You should neither get a plaintext when you have the digest beforehand nor should the digest provide its original value when you pass it through the hash function again.

3 Implementation

Initialize hash values:

```
private int[] H = {
        0x6a09e667,
        0xbb67ae85,
        0x3c6ef372,
        0xa54ff53a,
        0x510e527f,
        0x9b05688c,
        0x1f83d9ab,
        0x5be0cd19
};
```

Initialize array of round constants:

```
private static int[] K = {
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
        0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
        0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
        0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
        0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
        0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
        0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
};
```

Pre-processing (Padding):

1. begin with the original message of length L bits
2. append a single '1' bit
3. append K '0' bits, where K is the minimum number ¿= 0 such that (L + 1 + K + 64) is a multiple of 512
4. append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits such that the bits in the message are: 1 ¡L as 64 bit integer¿

```
private static byte[] preprocess(byte[] msgBytes) {
    int length = msgBytes.length;
    long bits = length*8;
    int msgLength = (length%64)*8;
    int padLength;
    //pad length is 512 - L = 1+k+64
    if ((512 - msgLength) > 8) {
        padLength = 512 - msgLength;
    } else {
        padLength = 1024 - msgLength;
    }
  //all bits in the pad are set to 0:
    byte[] pad = new byte[padLength/8];
  //append 1 bit to the pad:
    pad[0] = (byte) 0x80;
  //Length of message in 64 bits is appended (in Big Endian):
    for (int i = 0; i < 8; i++) {
     pad[pad.length - 1 - i] = (byte) (0xFF&(bits >>> (8 * i)));
    }
  //k bits are still 0.
    byte[] result = new byte[length + padLength/8];
    for (int i = 0; i<length; i++) {
     result[i]=msgBytes[i];
    }
    for (int i = length; i<pad.length+length; i++) {
     result[i]=pad[i-length];
```

```
        }
        return result;
    }
```

Process the message in successive 512-bit chunks:

```
    private static void msgProcessing() {
        for (int i=0; i<16; i++) {
            w[i] = 0;
            for (int j = 0; j<4; j++) {
                w[i] |= ((0x000000FF&chunk[j+4*i]) << (24-j*8));
            }
        }
        for (int i=16; i<64; i++) {
         w[i]=0;
            int s0 = Integer.rotateRight(w[i-15],7) ^ Integer.rotateRight(w[i-15],18) ^
                    (w[i - 15] >>> 3);
            int s1 = Integer.rotateRight(w[i-2],17) ^ Integer.rotateRight(w[i-2],19) ^
                    (w[i-2] >>> 10);
            w[i] = w[i-16] + s0 + w[i-7] + s1;
        }
    }
```

Compression function:

```
       private static void update(int[] h, int[] w, int j) {
    int S1 = Integer.rotateRight(h[4], 6) ^
               Integer.rotateRight(h[4], 11) ^
               Integer.rotateRight(h[4], 25);
    int ch = (h[4] & h[5]) ^ (~h[4] & h[6]);
    int temp1 = h[7] + S1 + ch + K[j] + w[j];

       int S0 = Integer.rotateRight(h[0], 2) ^
               Integer.rotateRight(h[0], 13) ^
               Integer.rotateRight(h[0], 22);
       int maj = (h[0] & h[1]) ^ (h[0] & h[2]) ^ (h[1] &
        h[2]);
       int temp2 = S0 + maj;

       h[7] = h[6];
       h[6] = h[5];
       h[5] = h[4];
       h[4] = h[3] + temp1;
       h[3] = h[2];
       h[2] = h[1];
       h[1] = h[0];
       h[0] = temp1 + temp2;
   }
```

Produce the final hash value (big-endian):

```
    public byte[] hash(byte[] msgBytes) {
       byte[] processed = preprocess(msgBytes);
       int[] hVals = new int[8];
       for (int i = 0; i<8; i++) {
        hVals[i]=H[i];
       }
       for (int i = 0; i < processed.length / 64; ++i) {
        for (int j = 0; j<processed.length; j++) {
        chunk[j]=processed[64*i+j];
        }
           msgProcessing();
           int[] temp = new int[8];
           for (int j = 0; j < 8; j++) {
            temp[j] = hVals[j];
           }
           for (int j = 0; j < 64; ++j) {
               update(temp, w, j);
           }
           for (int j = 0; j < 8; ++j) {
               hVals[j] += temp[j];
           }
       }
       byte[] finalHash = new byte[32];
       for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 4; j++) {
        finalHash[4*i+j]=toByteArray(hVals[i])[j];
        }
       }
       return finalHash;
   }
```

For the encryption/decryption was chosen RSA cipher. In the Runner class we take the user input message, store it in a database, generate hash value for the message, hash value together with a private key as input for RSA algorithm, encrypt and generate digital signature. After this we decrypt the recieved message and compare the two obtained results.

4 Conclusion

In conclusion, hashing is a useful tool to verify files are copied correctly between two resources. It can also be used to check if files are identical without opening and comparing them.