CHAPTER 12

Cryptographic Hash Functions

(Solution to Odd-Numbered Problems)

Review Questions

1. A cryptographic hash function takes a message of arbitrary length and creates a message digest of fixed length.

3. The Merkle-Damgard scheme is an iterated hash function that is collision resistant if the compression function is collision resistant. If we use this scheme, we need only to make the compression function collision resistant.

5. An iterated cryptographic hash function can use a symmetric-key block cipher as a compression function. We mentioned Rabin, Davies-Meyer, Matyas-Meyer-Oseas, and Miyaguchi-Preneel schemes.

7. Whirlpool is an iterated cryptographic hash function, based on the Miyaguchi-Preneel scheme, that uses a symmetric-key block cipher in place of the compression function. The block cipher is a modified AES cipher that has been tailored for this purpose. The following table shows some characteristics of Whirlpool.

Characteristics	*Values*
Minimum message size	$2 < ^{256}$ bits
Block size	512 bits
Message digest size	512 bits
Number of rounds	10

Exercises

9. The size of the length field is 128 bit or 32 hexadecimal digits.

a. 0000 0000 0000 0000 0000 0000 0000 **03EB**

b. 0000 0000 0000 0000 0000 0000 0000 **2710**

c. 0000 0000 0000 0000 0000 0000 000**F** **4240**

11. We need to have |M| + |P| + 128) mod 1024 = 0 or |P| = (–|M| – 128) mod 1024.

a. |P| = (– |M| – 128) mod 1024 = (–5120 – 128) mod 1024 = 896

b. |P| = (–||M| – 128) mod 1024 = (–5121 – 128) mod 1024 = 895

c. |P| = (–||M| – 128) mod 1024 = (–6143 – 128) mod 1024 = 897

13.

a. In SHA-512, the last block, which is 1024 bits, consists of

X | padding | 128-bit message length

In which, X is the rightmost part of the message of (|M| mod 1024) bits. If two messages are the same, then X is the same, the padding section is the same, the message length value is the same. This means the last block is the same.

b. In Whirlpool, the last block, which is 512 bits, consists of

X | padding | 256-bit message length

In which, X is the rightmost part of the message of (|M| mod 256) bits. If two messages are the same, then X is the same, the padding section is the same, the message length value is the same. This means the last block is the same.

15. The compression function of SHA-512 can be compared to a Feistel cipher (or encryption cipher) of 80 rounds:

a. The initial digest in the compression function can be thought as the plaintext in the Feistel cipher.

b. The final digest in the compression function can be thought as the ciphertext in the Feistel cipher.

c. Each word in the compression function can be thought as the corresponding round key in the Feistel cipher.

17. If the *final adding* operation is removed from the SHA-512 compression function, then its structure is similar to Rabin scheme, which is subject to the meet-in-the-middle attack as discussed in the textbook.

19. In AES, the need for removing the third operation is to make the encryption and decryption inverse of each other. In Whirlpool, we use a cryptographic cipher to simulate a hash function. The cipher is used only as an encryption algorithm without the decryption algorithm. The hash function can have any structure without worrying about the inverse structure.

21. $ShL_{12}(x)$ means the left shifting of the argument by 12 bits or 3 hexadecimal digit.

Before Shifting:	1234 5678 ABCD 2345 3456 5678 ABCD 2468
After Shifting:	4567 8ABC D234 5345 6567 8ABC D246 8000

23. The three blocks differ only in the first digit (leftmost digit). For the first digits, Conditional (0001, 0010, 0011) = $(0011)_2 = 2_{16}$. For the rest of the block, it means applying the Conditional function to three digits of equal values, which results in the common digit. Therefore, we have

Result: `2234 5678 ABCD 2345 3456 5678 ABCD 2468`

25. Although this operation is available in most high-level languages, we write a routine for that. The following routine calls another routine, RotR(x), which rotates right only one bit. We assume that the word is stored in an array of 64 bits with the leftmost bit as the first element and the rightmost bit as the last element.

```
RotR (x, i)
{
        count ← 1
        while (count < i)
        {
                RotR(x)
                count ← count + 1
        }
        return x
}
RotR(x)
{
        temp ← x[64]
        j ← 63
        while (j > 1)
        {
                x[j + 1] ← x[j]
                j ← j − 1
        }
        x[1]← temp
        return x
}
```

27. We assume that words *x*, *y*, *z* are represented as arrays of 64 elements. The following routine shows how to find the result. We have use the if-else statement to show the conditional nature of the operation. The code can be shorter if we use the logical operators (AND, OR, and NOT).

```
Conditional (x, y, z)
{
        i ← 1
        while (i ≤ 64)
        {
                if (x[i] = 1)
                        result[i] ← y[i]
                else
                        result[i] ← z[i]
```

```
                    i ← i + 1
            }
            return result
}
```

29. We call the RotR(x, i) function we used in Exercise 25.

```
Rotate (x)
{
        x1 ← x    x2 ← x    x3 ← x
        result ← RotR(x1, 28) ⊕ RotR(x2, 34) ⊕ RotR(x3, 39)
        return result
}
```

31. This is the same as the previous example, except that we need to use the first eighty primes and then take the cubic root of them. The routine uses the same routine, Convert64 defined in the solution to Exercise 30.

```
CalcConstants ()
{
        Primes [80] = {2, 3, …, 401, 409}
        i ← 1
        while (i ≤ 80)
        {
                temp ← (Primes [i])^1/3
                temp ← ExtractFraction (temp)
                Constants [i] ← Convert64 (temp)
                i ← i + 1
        }
        return (Constants[1 … 80])
}
```

33.

```
CompressionFunction (H[1 … 8], W[0 …79], K[0 … 79])
{
        i ← 1
        while (i ≤ 8)
        {
                Temp[i] ← H[i]
                i ← i + 1
        }
        j ← 0
        while (j < 80)
        {
                H[1 … 8] ←  RoundFunction(H[1 … 8], W[j], K[j])
                j ← j + 1
        }
        i ← 1
        while (i ≤ 8)
        {
                H[i] ← (H[i] + Temp[i]) mod 2^64
                i ← i + 1
        }
        return (H[1 … 8])
}
RoundFunction (H[1 … 8], W, K)
{
        i ← 1
        while (i ≤ 8)
        {
                T[i] ← H[i]
                i ← i + 1
        }
        H[2] ← T[1]
        H[3] ← T[2]
        H[4] ← T[3]
        H[6] ← T[5]
        H[7] ← T[6]
        H[8] ← T[7]
        Temp1 ← (Majority (T[1], T[2], T[3]) + Rotate [T[1]) mod 2^64
        Temp2 ← (Conditional (T[5], T[6], T[7]) + Rotate [T[5] + W + K) mod 2^64
        H[1] ← (Temp1 + Temp2) mod 2^64
        H[5] ← (Temp2 + T[4]) mod 2^64
```

```
        return (H[1 … 8])
}
```

35.

```
TransformStateToBlock (S[0 … 7][0 … 7])
{
        i ← 0
        j ← 0
        while (i < 8)
        {
                while (j < 8)
                {
                        b[i × 8 + j] ← S[i][j]
                        j ← j + 1
                }
                i ← i + 1
        }
        return (b[0 … 63])
}
```

37.

```
ShiftColumns (S[0 … 7] [0 … 7])
{
        c ← 1
        while (c ≤ 7)
        {
                shiftcolumn (S[c], c)
                c ← c + 1
        }
        return (S[0 … 7][0 … 7])
}
shiftcolumn (col[1 … 8], n)
{
        CopyColumn (col, temp)
        r ← 0
        while (r ≤ 7)
        {
                col [(r − n) mod 8] ← temp [r]
                r ← r + 1
        }
```

```
        return col[1 … 8]
}
```

39.

```
AddRoundKey (S[0 … 7][0 … 7], k[0 … 7][0 … 7])
{
        i ← 0
        j ← 0
        while (i < 8)
        {
                while (j < 8)
                {
                        s[i][j] ← s[i][j] ⊕ k[i][j]
                        j ← j + 1
                }
                i ← i + 1
        }
        return (S[0 … 7][0 … 7])
}
```

41.

```
RoundConstant ()
{
        r ← 0
        i ← 0
        j ← 0
        while (r ≤ 10)
        {
                while (i < 8)
                {
                        while (j < 8)
                        {
                                if (i = 0)
                                        RC[r][i][j] ← ByteTrans (8 × (r −1) + j)
                                else
                                        RC[r][i][j] ← 0
                                j ← j + 1
                        }
                        i ← i + 1
                }
                r ← r + 1
```

```
        }
        return (RC[0 … 10][0 … 7] [0 … 7])
}
```

43.

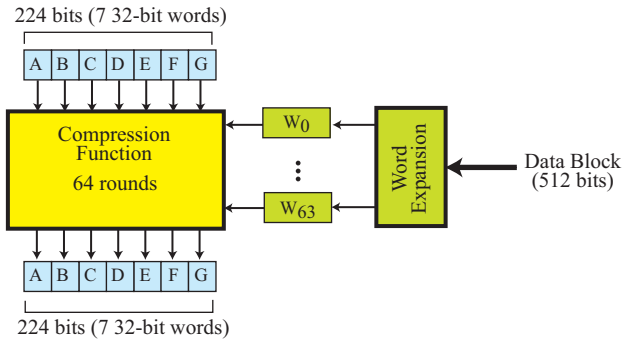
```
WhirlpoolHashFunction (M[1 … N], N)
{
        H ← (00 … 0)                          // 520 of 0's
        i ← 0
        while (i ≤ N)
        {
                H ← WhirlpoolCipher (M[i], H) ⊕ H ⊕ M[i]
                i ← i + 1
        }

        return H
}
```

45. All of these three hash functions are either similar to SHA-1 or SHA-512. The differences are in the block size, digest size, word size, and the number of rounds.

a. SHA-224 is very similar to SHA-1 (See Figure S12.45a) except that the digest size is 224 bits (7 words, each of 32 bits). The number of rounds is 64.

Figure S12.45a *Solution to Exercise 45 part a*



b. SHA-256 is very similar to SHA-1 (See Figure S12.45b) except that the digest size is 256 bits (8 words, each of 32 bits). The number of rounds is 64.

c. SHA-256 is very similar to SHA-512 (See Figure S12.45c) except that the digest size is 384 bits (6 words, each of 64 bits). The number of rounds is 80.

Figure S12.45b *Solution to Exercise 45 part b*

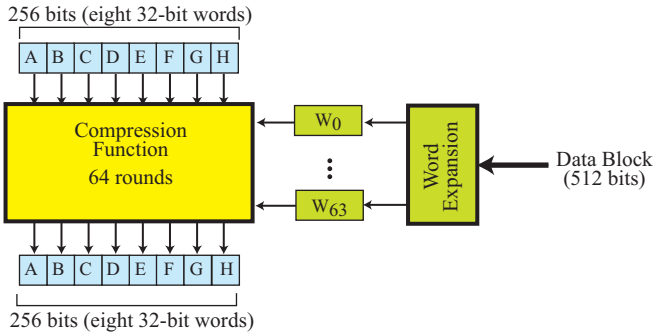
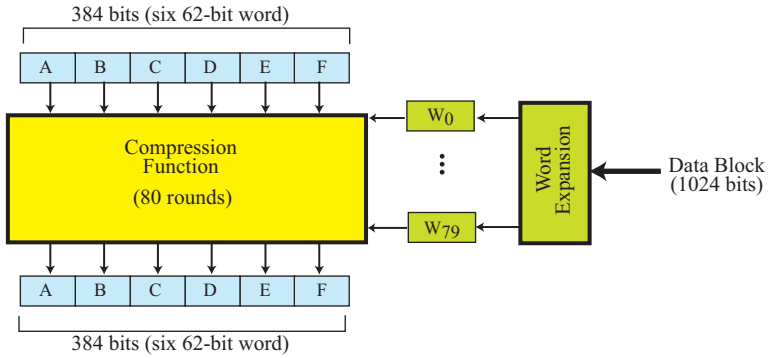


Figure S12.45c *Solution to Exercise 45 part c*



47.

a. HAVAL is a hashing algorithm of variable-size digest designed by Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry in 1992 as shown in Figure S12.47a.

The digest can be 128, 160, 192, 224, or 256 bits. The block size is 1024 bits. The algorithm actually creates a digest of 256 bits, but a folding algorithm matches the resulting 256 bits to one of the desired sizes.

b. The compression function uses 3, 4, and 5 passes in which each pass uses 16 iteration of different complex functions. The three-pass version is the fastest, but least secure; the five-pass version is the slowest, but the most secure. Figure S12.47b shows the structure of the compression function in HAVAL.

Figure S12.47a *Solution to Exercise 47 Part a*

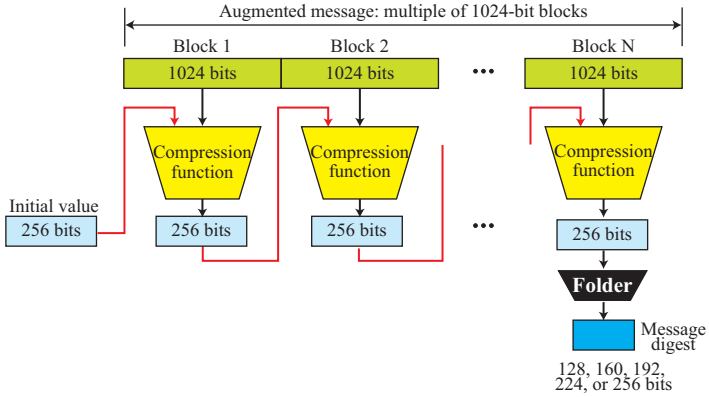


Figure S12.47b *Solution to Exercise 47 part b*