现代密码学作业 SHA3-256 实现文档

> 计26 郑睿阳

前言

- 本文档是关于清华大学计算机系 2024-春 现代密码学课程实现 SHA3-256 算法的作业。其不仅承担了对于程序的说明，也承担了对于 SHA3-256 介绍的作用，可供复习，学习时参考使用。

简介

- SHA3 是第三代安全散列算法的简称（注意，虽然同为 SHA 系列的算法，但实际上它的实现方法和 SHA-256 完全不同）。又名 Keccak 算法。

算法作用

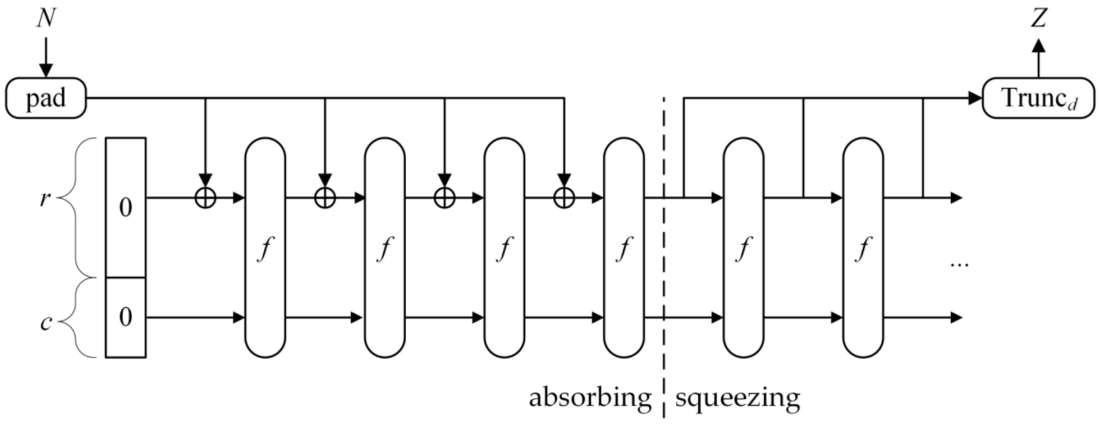
- 算法使用一个规模为 $5 * 5 * 64 = 1600$ bit 的密钥，接受的输入为任意长度，输出 32 字节的哈希值。

操作对象: State

- 我们通常使用 $w = 64$, 即 $5 * 5 * 64$ 的一个三维矩阵来表示状态。
- 定义 $A[x, y, z]$ 即为一个 bit 的三元组表示。

算法框架

- SHA3 的算法框架大致如下：
  - 明文填充 -> 多轮异或 + Keccak 函数 -> 挤压操作



明文填充

- 如上图，SHA3 算法采取了所谓的 “海绵结构” 作为设计。在加密的过程当中，经过 “吸收” 和 “挤压” 两个过程。其中，在吸收过程当中，需要将长度较长的明文进行分块，每一块都需要满足长度为上图中所定义的 r 的长度，并和密钥进行多次异或操作与 f 函数运算操作。这也就意味着，我们最后进行操作的明文总长度必须是 r 的倍数。因此，明文填充操作的目的也就是使得明文满足这一点。

- 我们只考虑整字节填充的情况，则SHA3 的明文填充服从以下的规则：首先我们计算出应该填充的字节数。假设字符串的长度为 len, 且 r, len 都是以字节为单位的，那么经过简单的模运算可以得到需要填充的字节数为

  $$q = r - len(mod(r))$$

  而填充的规则则是：

Type of SHA-3 Function	Number of Padding Bytes	Padded Message
Hash	$q = 1$	$M \Vert 0\text{x}86$
Hash	$q = 2$	$M \Vert 0\text{x}0680$
Hash	$q > 2$	$M \Vert 0\text{x}06 \Vert 0\text{x}00 \ldots \Vert 0\text{x}80$

  其中 $\Vert$ 符号表示的是字符串的连接符号。

- 在代码实现当中，明文填充功能由 `padding` 函数执行，代码如下：

```
int padding(unsigned char* data, int len) {
    // In SHA3-256 standard, we pad the message with 10*1.
    // Following the rules of byte padding, first calculate q. note that the
rate r used in this calculation stands for the number of bytes we use
    for (int i = 0; i < len; i++) {
        buf.push_back(data[i]);
    }
    int q = r - (len % r);
    if (q == 1) {
        buf.push_back((unsigned char)(0x86));
    }
    else if (q == 2) {
        buf.push_back((unsigned char)(0x06));
        buf.push_back((unsigned char)(0x80));
    }
    else {
        buf.push_back((unsigned char)(0x06));
        for (int i = 0; i < q - 2; i++) {
            buf.push_back((unsigned char)(0x00));
        }
        buf.push_back((unsigned char)(0x80));
    }
    return buf.size();
}
```

异或操作

- 通过上面的示意图，可以看到，我们会将填充过后的明文以 r 为块规模分成若干块。使用海绵结构的 “吸收” 功能实现初步加密。对于该功能在本文档当中不做说明，实际上上面图中的解释已经足够直观。

Keccak - f 函数

- 该函数即为上面的 f 函数。其内部执行的操作如下：

```
void keccakF() {
        for (int i = 0; i < 24; i++) {
            // Impl 5 functions
            theta();
            Rou();
            Pi();
            Chi();
            IotA(i);
        }
    }
```

对于一般的 SHA3 算法，keccak-f 函数的基本结构是，执行 n 轮的循环操作，每轮操作都需要按照顺序经过 $\theta, \rho, \pi, \chi, \iota$ 五个函数。

- **θ 函数**
- θ 函数接受状态矩阵 A 作为输入, 返回另一个状态矩阵 A'。
- 步骤如下:

Steps:

1. For all pairs (x, z) such that $0 \le x < 5$ and $0 \le z < w$, let
$$C[x, z] = \mathbf{A}[x, 0, z] \oplus \mathbf{A}[x, 1, z] \oplus \mathbf{A}[x, 2, z] \oplus \mathbf{A}[x, 3, z] \oplus \mathbf{A}[x, 4, z].$$
2. For all pairs (x, z) such that $0 \le x < 5$ and $0 \le z < w$ let
$$D[x, z] = C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod w].$$
3. For all triples (x, y, z) such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let
$$\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus D[x, z].$$

- 代码实现如下：

```
void theta() {
    uint64 tmp_state[5][5];
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            tmp_state[i][j] = state[i][j];
        }
    }
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            state[i][j] = tmp_state[i][j] ^ C((i + 4) % 5, tmp_state) ^ rotl(C((i
+ 1) % 5, tmp_state), 1);
        }
    }
}
```

ρ 函数

- ρ 函数接受状态矩阵 A 作为输入, 返回另一个状态矩阵 A'。
- 步骤如下：

Algorithm 2: ρ(**A**)

Input:
state array **A**.

Output:
state array **A′**.

Steps:

1. For all z such that $0 \le z < w$, let $\mathbf{A}'[0,0,z] = \mathbf{A}[0,0,z]$.
2. Let $(x,y) = (1,0)$.

---

3. For t from 0 to 23:
   a. for all z such that $0 \le z < w$, let $\mathbf{A}'[x,y,z] = \mathbf{A}[x,y,(z-(t+1)(t+2)/2) \bmod w]$;
   b. let $(x,y) = (y,(2x+3y) \bmod 5)$.
4. Return **A′**.

- 实际上，在算法的实现当中，ρ 函数可以理解为是一个沿着 z 方向的循环移位函数，坐标为 (x,y) 处的向量的移位数可以通过查表来得到。
- 代码实现如下：

```
void Rou() {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            state[i][j] = rotl(state[i][j], (rho_table[i][j]%64));
        }
    }
}
```

其中，模 64 的原因是因为 `rotl` 函数只接受移位数在 64 以内的移位数量（这与它使用位运算的实现方式有关）。而移位 n 位和移位 $n\ mod(64)$ 位的效果是一样的。

- **π 函数**
- π 函数接受状态矩阵 A 作为输入，返回另一个状态矩阵 A'。

- 步骤如下：

Algorithm 3: $\pi(\mathbf{A})$

Input:
state array **A**.

Output:
state array $\mathbf{A}'$.

Steps:

1. For all triples (x, y, z) such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let
   $\mathbf{A}'[x, y, z] = \mathbf{A}[(x + 3y) \bmod 5, x, z]$.
2. Return $\mathbf{A}'$.

- 该函数的一个直观理解是，可以认为它诱导了一个关于 x 坐标上的置换。具体而言，置换可以用图直观的表示如下：

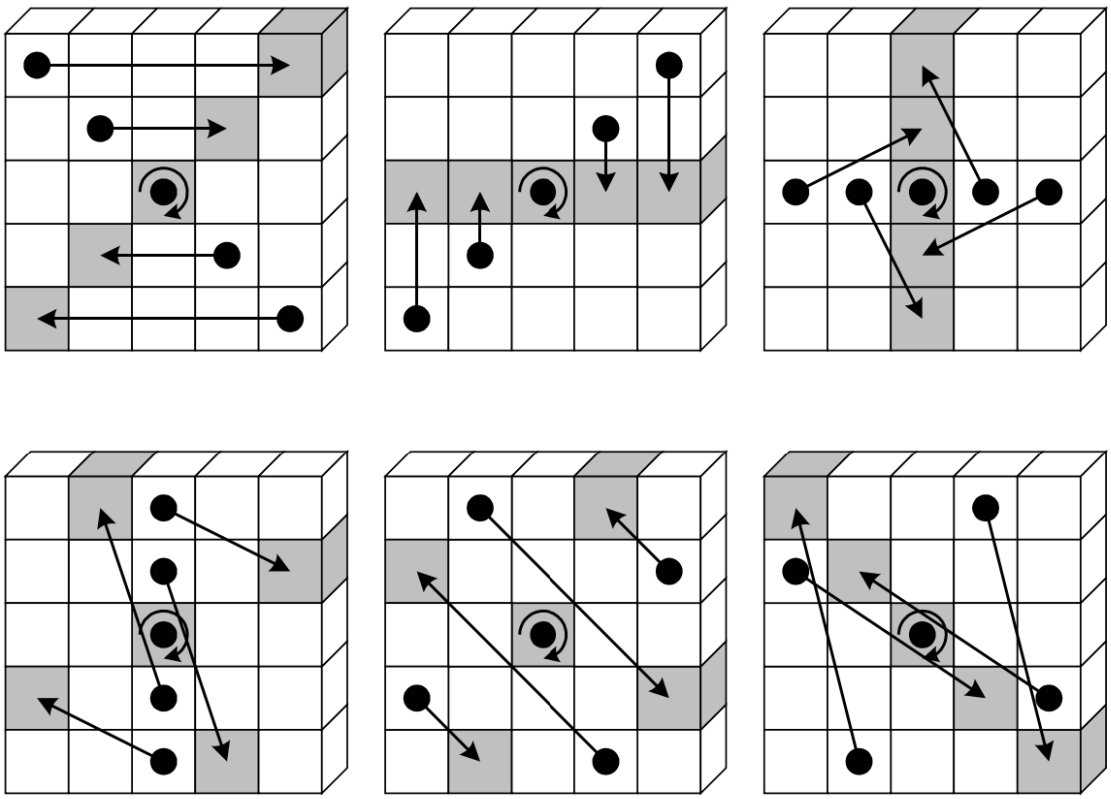


Figure 5: Illustration of π applied to a single slice [8]

- 可以看到的是，除了最中央的向量位置未发生变化之外，其它 24 个向量的全部变换到了 别的地方。那么，我们可以类比走“华容道”的思想，将一个向量取出，然后逐个按链式移动剩下的向量，这样就可以用较小的空间复杂度和时间复杂度实现算法。

- 实现的代码如下：

```
void Pi() {
    // We can also build a pi table ahead of time
    uint64 tmp = state[1][0];
    int this_index = 1, next_index = pi_table[1][0];
    while (next_index != 1) {
        state[this_index % 5][this_index / 5] = state[next_index % 5][next_index /
5];
        this_index = next_index;
        next_index = pi_table[this_index % 5][next_index / 5];
    }
    state[this_index % 5][this_index / 5] = tmp;
}
```

χ 函数

- χ 函数接受状态矩阵 A 作为输入, 返回另一个状态矩阵 A'。

- 步骤如下：

3.2.4 Specification of χ

Algorithm 4: χ(**A**)

Input:
state array **A**.

Output:
state array **A′**.

Steps:
1. For all triples (x, y, z) such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let
$$\mathbf{A}'[x,y,z] = \mathbf{A}[x,y,z] \oplus ((\mathbf{A}[(x+1) \bmod 5, y, z] \oplus 1) \cdot \mathbf{A}[(x+2) \bmod 5, y, z]).$$
2. Return **A′**.

- 代码实现如下：

```
void Chi() {
    uint64 tmp_state[5];
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            tmp_state[j] = state[j][i];
        }
        for (int j = 0; j < 5; j++) {
            state[j][i] = tmp_state[j] ^ (~tmp_state[(j + 1) % 5] & tmp_state[(j +
2) % 5]);
        }
    }
}
```

ι 函数

- ι 函数接受状态矩阵 A 以及迭代轮数作为输入, 返回另一个状态矩阵 A'。
- 步骤如下:

Algorithm 5: $rc(t)$

Input:
integer t.

Output:
bit $rc(t)$.

Steps:

1. If t mod 255 = 0, return 1.
2. Let $R = 10000000$.
3. For i from 1 to t mod 255, let:
    a. $R = 0 \| R$;
    b. $R[0] = R[0] \oplus R[8]$;
    c. $R[4] = R[4] \oplus R[8]$;
    d. $R[5] = R[5] \oplus R[8]$;
    e. $R[6] = R[6] \oplus R[8]$;
    f. $R = \text{Trunc}_8[R]$.
4. Return $R[0]$.

Algorithm 6: $\iota(\mathbf{A}, i_r)$

Input:
state array $\mathbf{A}$;
round index i_r.

Output:
state array $\mathbf{A}'$.

Steps:

1. For all triples (x, y, z) such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z]$.
2. Let $RC = 0^w$.
3. For j from 0 to ℓ, let $RC[2^j - 1] = rc(j + 7i_r)$.
4. For all z such that $0 \le z < w$, let $\mathbf{A}'[0, 0, z] = \mathbf{A}'[0, 0, z] \oplus RC[z]$.
5. Return $\mathbf{A}'$.

- 实际上也可以通过查表来实现：

```
void IotA(int round) {
    state[0][0] ^= RC[round];
}
```

其中，`RC` 数组是预先计算好的数组。可以根据上面的算法进行计算。

程序运行情况

- 为了保证程序的效率，我们需要对于程序开 `-O2` 优化：

```
g++ -O2 SHA3-256.cpp -o SHA3-256.exe
```

我们在 wsl 下运行我们的程序。其中，wsl 的版本号如下：

```
WSL 版本： 2.1.5.0
内核版本： 5.15.146.1-2
WSLg 版本： 1.0.60
MSRDC 版本： 1.2.5105
Direct3D 版本： 1.611.1-81528511
DXCore 版本： 10.0.25131.1002-220531-1700.rs-onecore-base2-hyp
Windows 版本： 10.0.22631.3527
```

我们分别使用 $8k$ 和 $8M$ 字节的字节流来测试程序的加密能力：

在 $8k$ 字节的程序当中由于速度过快无法准确的测出加密的速度：

```
zhengry22@zhengruiyang:/mnt/d/Cryptography/Cryptography/SHA3-256$ ./SHA3-256.exe 8kbit_data.txt output.txt
 inputfile: 8kbit_data.txt outputfile: output.txt
the hash value is :
f6 d8 27 0d 48 d7 16 a2 04 0a 17 cb ab 62 23 18 24 06 b5 f0 3e 3a 6d 9b d4 e8 96 2a ef 59 e0 dc
SHA3 256 Use: 0.12ms
The Speed is: 0Mbps
```

在 $8M$ 字节的程序当中可以测算出程序运行的速度为 $114Mbps$:

```
zhengry22@zhengruiyang:/mnt/d/Cryptography/Cryptography/SHA3-256$ ./SHA3-256.exe 8mbit_data.txt output.txt
 inputfile: 8mbit_data.txt outputfile: output.txt
the hash value is :
ca 87 fc c2 da ad b2 5d 76 89 08 39 d7 45 30 d7 67 d0 ea 8e 90 8c d1 f7 60 86 3c ff 50 e3 a3 e8
SHA3 256 Use: 70.128ms
The Speed is: 114.077Mbps
zhengry22@zhengruiyang:/mnt/d/Cryptography/Cryptography/SHA3-256$ ./SHA3-256.exe 8mbit_data.txt output.txt
 inputfile: 8mbit_data.txt outputfile: output.txt
the hash value is :
ca 87 fc c2 da ad b2 5d 76 89 08 39 d7 45 30 d7 67 d0 ea 8e 90 8c d1 f7 60 86 3c ff 50 e3 a3 e8
SHA3 256 Use: 70.187ms
The Speed is: 113.981Mbps
zhengry22@zhengruiyang:/mnt/d/Cryptography/Cryptography/SHA3-256$ ./SHA3-256.exe 8mbit_data.txt output.txt
 inputfile: 8mbit_data.txt outputfile: output.txt
the hash value is :
ca 87 fc c2 da ad b2 5d 76 89 08 39 d7 45 30 d7 67 d0 ea 8e 90 8c d1 f7 60 86 3c ff 50 e3 a3 e8
SHA3 256 Use: 69.878ms
The Speed is: 114.485Mbps
zhengry22@zhengruiyang:/mnt/d/Cryptography/Cryptography/SHA3-256$ ./SHA3-256.exe 8mbit_data.txt output.txt
 inputfile: 8mbit_data.txt outputfile: output.txt
the hash value is :
ca 87 fc c2 da ad b2 5d 76 89 08 39 d7 45 30 d7 67 d0 ea 8e 90 8c d1 f7 60 86 3c ff 50 e3 a3 e8
SHA3 256 Use: 69.986ms
The Speed is: 114.309Mbps
```

参考资料

清华大学 2024春 《现代密码学》课件

SHA3 官方文档：SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions