密码学第九次实验报告

SHA-1 哈希算法

原理

SHA-1 哈希算法是比较经典的一种哈希算法, 适用于数据的验证等需求. 哈希算法是单向算法, 给定原来的数据, 能够产生长度固定的哈希值.

SHA-1 哈希算法采用了 Merkle-Damgård 结构, 这种结构比较简单, 在理论上也能证明: 只要压缩函数 f 满足无碰撞性, 整个哈希函数也满足无碰撞性. 首先把消息按一定的规则进行填充, 然后把填充后的分组按顺序放入 f 函数, 输出作为下一轮执行 f 函数的参数之一. 第一轮没有上一轮执行 f 函数的结果, 就用初始向量 IV 代替. 其实, SHA-1 和 MD5 也比较相似.

SHA-1 哈希算法的 f 函数是通过不同的混淆和置换方式, 来达到相对安全的消息杂凑的. 其中有模 2^{32} 加法、按位运算等方式.

实际上, SHA-1 已经能够相对容易地产生碰撞了, 并且由于硬件计算能力的日益强大和算法的进步, 攻击正在变得越来越现实. 因此, SHA-1 已经可以认为被废弃了.

伪代码

算法 1: SHA-1 哈希算法 f 函数

输入: 数据块 b，上次的结果 l

输出: 结果 r

for $i =$ *0 ... 15* **do**
  $w_i \leftarrow b$ 的第 $4i$ 个字节到第 $4i+3$ 个字节按大端序转换成的整数
end
for $i =$ *16 ... 79* **do**
  $w_i \leftarrow (w_{i-3} \oplus w_{i-8} \oplus w_{i-14} \oplus w_{i-16}) <<< 1$
end
for $i =$ *0 ... 4* **do**
  $h_i \leftarrow l$ 的第 $4i$ 个字节到第 $4i+3$ 个字节按大端序转换成的整数
end
$a, b, c, d, e \leftarrow h_0, h_1, h_2, h_3, h_4$
for $i =$ *0 ... 79* **do**
  if $0 \le i \le 19$ **then**
    $f \leftarrow (b \wedge c) \vee ((\neg b) \wedge d)$
    $k \leftarrow$ `0x5a827999`
  end
  if $0 \le i \le 39$ **then**
    $f \leftarrow b \oplus c \oplus d$
    $k \leftarrow$ `0x6ed9eba1`
  end
  if $40 \le i \le 59$ **then**
    $f \leftarrow (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$
    $k \leftarrow$ `0x8f1bbcdc`
  end
  if $60 \le i \le 79$ **then**
    $f \leftarrow b \oplus c \oplus d$
    $k \leftarrow$ `0xca62c1d6`
  end
  $e, d, c, b, a \leftarrow d, c, b <<< 30, a, (a <<< 5) + f + e + k + w_i$
end
$h_0, h_1, h_2, h_3, h_4 \leftarrow h_0 + a, h_1 + b, h_2 + c, h_3 + d, h_4 + e$
return 各 h_i 按大端序转换成 *4* 个字节再按下标顺序拼接的字符串

算法 2: SHA-1 哈希函数消息分组函数

输入: 消息 m

输出: 填充后的消息 m'

$m' \leftarrow m$ 后面附加一个比特位 1

m' 后面不断填充比特位 0，直到其长度 l（以位为单位）满足

$l \equiv 448 (\bmod 512)$

m' 后面增加以 64 位大端序整数表示的 l

算法 3: Merkle–Damgård 结构哈希算法

输入: 初始向量 iv，填充后的消息 m'，哈希函数的 f 函数 $f(iv, m')$

输出: 哈希值 h

$result \leftarrow iv$

for $b = m'$ 的每一块 **do**

  $result \leftarrow f(result, b)$

end

return $result$

算法 4: SHA-1 算法

输入: 消息 m

数据: SHA-1 算法的初始向量 iv

$m' \leftarrow m$ 填充后的消息

$result \leftarrow$ Merkle–Damgård 结构哈希算法 (iv, m', f)

return $result$

HMAC 算法

算法 5: SHA-1 HMAC 算法

输入: 消息 m，密钥 k，哈希函数 $f(m)$，它的输出字节数 l_0

数据: 两个掩码字节串 $ipad, opad$

$l \leftarrow k$ 的字节数

if $l > l_0$ **then**

| $k \leftarrow f(k)$

end

$k \leftarrow k \parallel l_0 - l$ 个 0 字节

$k_1 \leftarrow k \oplus ipad$

$H_1 \leftarrow f(k_1 \parallel m)$

$k_2 \leftarrow k \oplus opad$

$H_2 \leftarrow f(k_2 \parallel H_1)$

return H_2

SHA-1 HMAC 算法

算法 6: SHA-1 HMAC 算法

输入: 消息 m，密钥 k

数据: SHA-1 哈希函数 f，SHA-1 哈希函数的输出字节数 l_0

return *HMAC* 算法 (m, k, f, l_0)

分析

以下设消息长度为 l.

f 函数

由于 f 函数的输入规模和计算语句的执行次数恒定, 所以它的时空复杂度都是 $\mathrm{O}(1)$.

消息分组函数

消息分组函数附加的数据长度是有上限的, 但是产生的分组与长度大致成正比关系. 因此易知它的时间复杂度为 $\mathrm{O}(l)$. 由于需要的空间上限是固定的 (最多 512 位), 空间复杂度为 $\mathrm{O}(1)$.

Merkle–Damgård 结构哈希算法

该结构的哈希算法的时间复杂度由 f 函数决定. 这里 f 函数时空复杂度都是 $\mathrm{O}(1)$. 但是分组有 $\mathrm{O}(l)$ 个, 所以时间复杂度是 $\mathrm{O}(l)$. 由于数据之间没有依赖, 而且 f 函数的空间复杂度是 $\mathrm{O}(1)$, 所以空间复杂度是 $\mathrm{O}(1)$.

SHA-1 算法

该算法就是 Merkle–Damgård 结构哈希算法的一个封装, 所以时间复杂度和空间复杂度也分别是 $\mathrm{O}(l)$ 和 $\mathrm{O}(1)$.

HMAC 算法

设密钥的长度为 l_K.

首先若密钥长度超过相应哈希算法的分组长度, 则密钥的哈希值就要被计算出来. 所以这里的时间复杂度是 $\mathrm{O}(l_K)$, 空间复杂度是 $\mathrm{O}(1)$. 之后, 密钥的长度就固定了. 对密钥的填充和与 $ipad$ 或 $opad$ 的异或时空复杂度都是 $\mathrm{O}(1)$.

然后对消息和密钥拼接, 再求哈希值. 这里密钥经过处理后, 长度恒定, 所以这步的时间复杂度是 $\mathrm{O}(l)$, 空间复杂度是 $O(1)$.

之后的步骤数据规模恒定, 需要的时间和空间也恒定, 所以时空复杂度都是 $\mathrm{O}(1)$.

所以算法总的时空复杂度分别是 $\mathrm{O}(l_K + l)$ 和 $\mathrm{O}(1)$.

SHA-1 HMAC 算法

该算法是 HMAC 算法的一个封装, 所以时空复杂度和它相同, 也都分别是 $\mathrm{O}(l_K + l)$ 和 $\mathrm{O}(1)$.

优化

f 函数

$i \in [0, 19]$ 时的逻辑函数是按位选择函数, $i \in [40, 59]$ 时的逻辑函数是按位取多数函数. 这些函数都有多种变形, 有的利于通用处理器实现, 有的利于专用电路实现.

w_i 的计算在第 32 - 79 轮时可以优化成 $w_i = (w_{i-6} \oplus w_{i-16} \oplus w_{i-28} \oplus w_{i-32}) <<< 2$. 这种变换可以使得各操作保持 64 字节对齐, 并且把 w_i 向 w_{i-3} 的依赖去掉了, 更有利于 SIMD 等向量指令集实现 SHA-1.

消息分组函数

消息分组函数可以计算出要填充的消息末尾后, 用 Python 的迭代器实现, 只要迭代到消息末尾后, 再追加即可. 这样可以减少中间结果的内存占用, 也可以在 HMAC 算法中复用.

Merkle-Damgård 结构哈希算法和 SHA-1 算法

这种算法虽然安全性较高, 但是好像数据依赖性也较高, 相对难以在并行性上优化. 但是可以通过优化 f 函数, 间接优化哈希算法.

HMAC 算法和 SHA-1 HMAC 算法

由于 $K^+ \oplus ipad$ 和 $K^+ \oplus opad$ 是可以提前计算出来的, 而且如果下层的哈希算法是 Merkle-Damgård 结构的话, 可以提前把这块放入 f 函数计算结果, 并且把结果当作相应的新初始向量 iv', 所以 SHA-1 HMAC 算法可以这样优化. 同样地, 并行性也是一个问题, 但是提前计算出新的 iv' 能够稍微提高并行性, 而且能够提高计算短消息的 HMAC 的速度.

测试

采用 `sha1_test.py` 进行自动测试. 该文件大致上是选择 10000 个随机字节串作为消息, 10000 个随机字节串作为密钥, 把求出的结果与标准库中

的 SHA-1 和 HMAC 算法进行比较. 如果有不同, 就打印出产生错误的消息和密钥, 否则打印出“`test passed`”并退出.

运行了该文件多次, 都能通过测试. 所以可以认为算法没有问题.

Hash 函数生日攻击

原理

对 Hash 函数的生日攻击是把 Hash 函数看成输入随机输出也随机但对某个确定输入输出确定的函数. 这种假设也符合理想 Hash 函数的性质. 这里的生日攻击是找出一对消息 x 和 y, $s.t. H(x) = H(y)$. 其中 $H(m)$ 为哈希函数.

可以通过概率论的知识得到, 若 Hash 函数是理想 Hash 函数, 攻击的代价是大约 $2^{\frac{n}{2}}$ 次 Hash 运算, 其中 n 是 Hash 函数结果的二进制位数.

伪代码

算法 7: 哈希函数生日攻击

输入: 随机字节串生成函数 $r()$，哈希函数 $H(m)$

输出: 一对字节串 x_1, x_2

$x_1, x_2 \leftarrow r(), r()$

while $H(x_1) \neq H(x_2)$ *or* $x_1 = x_2$ **do**

  $x_1, x_2 \leftarrow r(), r()$

end

return x_1, x_2

分析

由于攻击的代价是大约 $2^{\frac{n}{2}}$ 次 Hash 运算, 而且每次 Hash 运算的时空复杂度都是 $\mathrm{O}(1)$, 再加上每次 Hash 计算时没有数据依赖, 所以整个算法的时间复杂度为 $\mathrm{O}(2^{\frac{n}{2}})$, 空间复杂度为 $\mathrm{O}(1)$.

优化

其实该算法相当好并行, 所以可以并行计算, 这样可以线性地提高效率. 其实也可以先计算短消息, 再不断延长, 这样可以直接把已经算好的哈希作为新的初始向量, 提高计算效率.

实际上, 对已经有的哈希算法采用这种攻击不现实, 因为 n 太大了. 在真正测试的时候, 采用了截断的哈希函数, 也就是把消息经过 SHA-1 哈希算法的结果取前两个字节. 这样 $n = 16$, 能够保证在可行的时间内找到一对哈希值相同的消息.

测试

测试采用自动化测试, 文件为 `hash_collision_test.py`. 文件会自动找出一对有冲突的消息, 并输出它们的内容和冲突的哈希值. 这样实际上就起到了对算法进行测试的作用. 经过检验, 找到的几对有冲突的消息确实是有冲突.

英文消息变形生成算法

原理

对英文消息找出消息变形，有两种比较实际的方式：根据语义的方式和插入“空格—退格”对的方式。但是，根据语义的方式其实比较难，因为难以编写确定性算法。因此，采用了插入“空格—退格”对的方式。

插入“空格—退格”对的方式比较容易实现，但是一般是在空格附近插入“空格—退格”对。一般来说，需要用一种方式遍历原来消息中的空格，而且要注意消息中的空格数目应该是变化的。有一种可行的方式是用一个自然数表示，这样想的话，就变成了一个数学问题。设原来消息的空格数为 n，那么由于对每个空格前面（不妨设在每个空格前面插入“空格—退格”对）都可以插入自然数（这里也包括 0）个“空格—退格”对，所以这变成了找到一种从 N 到 N^n 的映射。可以证出 N^n 可数，即其势与 N 相等。因此

双射应该是能找到的。但是，考虑到实现的难易程度，这里只是构造了一个（或者说一系列）普通映射。

该映射如下：

$$
\begin{gathered}
f_n : \mathrm{N} \to \mathrm{N}^n \\
x \to (a_0, a_1, \cdots, a_{n-1})
\end{gathered}
$$

其中 $a_{x \mod n} = \lfloor \frac{x}{n} \rfloor + 1$，其它各 $a_i = \lfloor \frac{x}{n} \rfloor$。这样，就能利用这个映射实现能够意识到原来数据空格个数的消息变形，效果比较好。

由于需要变形的消息是英文消息，所以把英文消息表示成使用 8 位 ASCII 码的字节串形式。这样方便解释算法，也方便实现。

伪代码

算法 8: 英文消息变形生成算法

输入: 消息 m，要得到的变形数目 n

输出: 消息 m 的 n 个变形组成的列表（按一定顺序排列）

数据: 上一节中提到的映射 f

$result \leftarrow$ 空列表

$c \leftarrow m$ 中的空格数目

if $c = 0$ **then**

  抛出错误

end

for i = *1* ... n **do**

  $\vec{a} \leftarrow f_c(i)$

  for j = *1* ... c **do**

    $curr \leftarrow$ 在 m 中的第 j 个空格前面加入 a_j 个“空格—退格”对得到的结果

  end

  在 $result$ 后面添加 $curr$

end

分析

设原消息长度为 l 字节，空格个数为 n，要生成的变形个数为 c。

首先，根据 $f_n(x)$ 映射的性质，能知道最终列表中每个消息的长度不会超过 $l+2\lceil\frac{c}{n}\rceil$ 字节。而且，最长的那些消息的长度也是这个数量级。然后，要生成的变形个数为 c，因此算法的总空间复杂度为 $\mathrm{O}(cl+\frac{c^2}{n})$。

然后，对消息遍历一遍来找空格个数的时间复杂度为 $\mathrm{O}(l)$。对每个消息，插入空格的时间复杂度为 $\mathrm{O}(n)$。但是一共有 c 个消息，而且对每个变形消息，隐含拷贝消息的步骤，每次拷贝时间复杂度为 $\mathrm{O}(l+n)$。因此，算法的总时间复杂度为 $\mathrm{O}(cl+cn)$。

优化

首先很容易想到的一点，就是优化插入空格。对插入空格的优化，可以通过链表来进行，因为链表是一种插入时时空复杂度都是 $\mathrm{O}(1)$ 的线性表。而且，只要边读边检测空格时再加上建立链表的步骤，建立链表的代价就不大重要了。

然后，可以来优化映射，把插入得少的一些位用上，这样来优化时空复杂度。首先，N 到 N^2 的双射可以利用对角线方法得到，因此更高维度的双射，可能也可以用类似方法得到。

事实上，其实还有其它同学设计的算法（或者也可以说映射）比这个映射更好，在这里就不赘述了。

测试

采用自动化脚本进行测试。测试脚本文件名为 `msg_permutation_test.py`。脚本主要通过一个英文短语，测试能否得到消息变形列表，以及该列表的正确性。经过测试，发现能够得到消息列表，该列表也符合前面定义的规则。所以可以说，实现一般是正确的。

SHA-3 哈希算法

原理

SHA-3 哈希算法是为了取代 SHA-2 哈希算法而被创造的，它能提供比 SHA-2 更强的安全性。它的核心并不是分组链接结构，而是海绵结构，这种结构是一种全新的结构，它能够提供更高的灵活性。类似 AES，SHA-3 哈希算法也是被公开评选出来的，这使得加密算法的选择更为透明，也比较能避免算法中的后门。

被公开评选出来的 SHA-3 哈希算法是 Keccak 系列密码算法中的一种，它由同名的 Keccak 团队发明出来，经过了比较先进的安全强度检验。Keccak 系列密码算法的核心是 f 函数，它也决定了 Keccak 系列算法中间状态的大小。中间状态是三维的、分行（row）、列（column）和纵（lane）。其中行和列的数量固定为 5。它的核心是五个按顺序执行的变换：$\theta, \rho, \pi, \chi, \iota$ 变换。它们比较完善地践行了对称密码体制的两个基本变换：代替和置换。它们有的负责一个纵内的变换，有的负责行和列的变换；有的负责进行线性变换，有的负责进行非线性变换；有的负责改变纵的位置，有的负责改变纵的内容；有的保持对称性，有的引入常数来打破对称性。综合来看，这是一种理论坚实、算法完善、经得起检验的密码算法。而且更为可贵的是，它的标准是开放的，Keccak 团队甚至在他们的网站（`https://keccak.team`）上公布了标准。

SHA-3 哈希算法就是在 f 函数上进行的。但是，它用到的是海绵结构。海绵结构有两个阶段：吸收阶段吸收输入的信息，挤压阶段处理（也可以看成放出）输入的信息。通过保持 f 函数的输入比特率和容纳量之和不变（因此 SHA-3 系列哈希算法只用到一种 f 函数），更改输入比特率和输出长度，SHA-3 哈希算法能做到输出可变长度的哈希值，在这方面做到了在各个方面能够替代原先的 SHA 算法的要求。

这里用到的 SHA-3 哈希算法，是遵从 Keccak 团队官方网站上的标准的。这样，能够调用 `hashlib` 库进行自动化测试。

值得注意的一点是，在把状态打包成字节时，应该按照列优先的方式进行打包，而且把每个纵按照小端序的方式进行打包。在解包时，也要做与其对应的操作。这样才能构造出符合正确的 SHA-3 算法。这样，也能把五

个原来基于位变换的操作，转换成基于整个纵的变换，提高了效率。

基于 SHA-3 的 HMAC 算法和普通的 HMAC 算法没有什么区别，不过按照标准，每块的长度需要指定。因此，优化会麻烦一些。

伪代码

算法 9: SHA-3 哈希算法 f 函数

输入: 200 个字节组成的字节串 b

输出: 变换后的字节串 b'

数据: 循环移位位数表 $r[0\cdots4][0\cdots4]$，轮常数 $rc[0\cdots23]$

```
把 b 按照前面的方式转换，得到 64 位整数数组 A[0···4][0···4]
for i = 0 ... 23 do
    for x = 0 ... 4 do
        C[x] ← A[x,0] ⊕ A[x,1] ⊕ A[x,2] ⊕ A[x,3] ⊕ A[x,4]
    end
    for x = 0 ... 4 do
        D[x] ← C[(x − 1) mod 5] ⊕ (C[(x + 1) mod 5] <<< 1)
    end
    for x = 0 ... 4 do
        for y = 0 ... 4 do
            A[x,y] ← A[x,y] ⊕ D[x]
        end
    end
    for x = 0 ... 4 do
        for y = 0 ... 4 do
            B[y][(2x + 3y) mod 5] ← A[x][y] <<< r[x][y]
        end
    end
    for x = 0 ... 4 do
        for y = 0 ... 4 do
            A[x][y] ← B[x][y] ⊕ ((¬B[(x + 1) mod 5][y]) ∧ B[(x + 2)
              mod 5][y])
        end
    end
    A[0][0] ← A[0][0] ⊕ rc[i]
end
return A[0···4][0···4] 按照对应的方式转换回来的字节串 b′
```

算法 10: Keccak 哈希函数构造主算法

输入: 消息 m，比特速率 r，1 个字节长的限定后缀 d，输出长度（以字节为单位）l

输出: 长度为 l 字节的哈希值 h

数据: 纵长度 w

$P \leftarrow m \parallel d$

在 P 后面附加 0 字节，使得其长度（以字节为单位）能被 $\lfloor \frac{r}{8} \rfloor$ 整除

P 的最后一个字节 $\leftarrow$ 它与 `0x80` 异或后的结果

$b \leftarrow$ 200 字节长的字节数组

for P 中每个 $\lfloor \frac{r}{8} \rfloor$ 字节大小的块 **do**

  b 的前 $\lfloor \frac{r}{8} \rfloor$ 个字节 $\leftarrow$ 它与该块逐字节异或后的结果

  $b \leftarrow f(b)$

end

$h \leftarrow$ 空字节串

while 还需要更多的输出 **do**

  $h \leftarrow h \parallel b$

  $b \leftarrow f(b)$

end

算法 11: SHA-3 哈希算法

输入: 需要的哈希值位数 l，消息 m

输出: 长度为 $\lfloor \frac{l}{8} \rfloor$ 字节的哈希值 h

if $l = 224$ **then**
　return $Keccak(m, 1152, \texttt{0x06}, 224)$
end
if $l = 256$ **then**
　return $Keccak(m, 1088, \texttt{0x06}, 256)$
end
if $l = 384$ **then**
　return $Keccak(m, 832, \texttt{0x06}, 384)$
end
if $l = 512$ **then**
　return $Keccak(m, 576, \texttt{0x06}, 512)$
end
抛出错误

算法 12: SHA-3 HMAC 算法

输入: 需要的 MAC 值位数 l，消息 m，密钥 k

输出: MAC 值 h

数据: 基于哈希函数 $f(m)$ 的 HMAC 算法 $\mathrm{HMAC}(m, k, f, l_O)$,
SHA-3 哈希函数 $f(l, m)$

if $l = 224$ **then**
  $g(m) \leftarrow f(224, m)$
  return $\mathrm{HMAC}(m, k, g, 224)$
end
if $l = 256$ **then**
  $g(m) \leftarrow f(256, m)$
  return $\mathrm{HMAC}(m, k, g, 136)$
end
if $l = 384$ **then**
  $g(m) \leftarrow f(384, m)$
  return $\mathrm{HMAC}(m, k, g, 104)$
end
if $l = 512$ **then**
  $g(m) \leftarrow f(512, m)$
  return $\mathrm{HMAC}(m, k, g, 72)$
end
抛出错误

分析

设消息长度为 l，密钥长度为 l_k，哈希算法的输出长度为 l_o。

SHA-3 哈希算法 f 函数

易知算法空间上界和计算上界都固定，因此它的时空复杂度都是 $\mathrm{O}(1)$。

Keccak 哈希函数构造主算法

首先，在初始化阶段，需要 $\mathrm{O}(l)$ 量级的空间来存储 P，但是生成 P 一般需要 $\mathrm{O}(1)$ 的时间复杂度。然后，易知分块数目在 $\mathrm{O}(l)$ 量级。这样，由于对每个分块都需要执行一次 f 函数，所以吸收阶段的时间复杂度为 $\mathrm{O}(l)$，空间复杂度也为 $\mathrm{O}(l)$。之后，在挤压阶段，易知每次输出的块长度固定。而每次挤压需要的时空复杂度都为 $\mathrm{O}(1)$。因此，挤压阶段需要的时空复杂度都是 $\mathrm{O}(l_o)$。算法总时空复杂度都是 $\mathrm{O}(l + l_o)$。

SHA-3 哈希算法

易知该算法就是 Keccak 哈希函数构造主算法的一个封装，因此时空复杂度与 Keccak 哈希函数构造主算法相同，都是 $\mathrm{O}(l + l_o)$。

SHA-3 HMAC 算法

易知该算法是 HMAC 算法的一个封装，而第一次求哈希值的时空复杂度根据上面和 HMAC 算法的步骤知这一步的时空复杂度都是 $\mathrm{O}(l_k + l_o)$，因为第一次求哈希值可能还包括把密钥 k 再求一遍哈希值的步骤。之后，求第二遍哈希值时，时空复杂度由于指定的哈希值字节数给定，为 $\mathrm{O}(1)$。因此，算法的总时空复杂度都是 $\mathrm{O}(l_k + l_o)$。

优化

SHA-3 哈希算法 f 函数

实际上，这个函数的每一步变换，原来是基于位的，而且是把状态作为比特串做的运算。但是，根据位运算的性质，可以把每个纵看成一个整体，把它变成每个纵整体的位运算。这样既能提高效率，也能方便实现。

还有一点，就是 ρ 变换的移位次数不要现算，可以查表，因为现算太麻烦了。这样显然能提高效率，是一种以时间换空间的做法。

Keccak 哈希函数构造主算法

该算法主要的优化点就是海绵结构。首先可以不实际上合并 m 和 d，只建立对它们的引用即可，而且应该是只维护两项，这样能够省下空间。然后，可以只维护 P 的偏移，看到剩下的空间不足一个块了，就现场填充和异或在一起做（实际上也就不用填充太多了，因为填充数据是 0 字节），这样也能够省下空间，减少最后一次异或的个数。

最后，由于输出空间可以预知，所以可以提前分配，挤压阶段中得到每一步的输出，检查还剩多少字节，并把相应的结果写入已经分配好的连续空间内即可。这样能够利用内存访问的连续性提高效率，也能减少分支数量，间接提高效率。

SHA-3 哈希算法

该算法主要的优化点是哈希所用参数的查找。当然，也可以通过某些数学关系，不需要查找，直接计算出需要的参数。不过，最容易实现的还是查找表。

SHA-3 HMAC 算法

该算法相对其它 HMAC 算法的一个独特之处就是它把哈希函数的输出字节数改变了，而且都比原来相应哈希函数的输出字节数长。这样实现能够符合 Python 中 `hashlib` 的结果。因此，一个比较明显的优化点是优化第一块的计算。类似基于分组链接结构的哈希算法改变初始向量 iv，这里也可以提前计算出状态 b。不过由于长度的差异，优化时可能要注意特殊情况。

测试

采用自动化脚本进行测试，脚本文件名为 `sha3_test.py`。该脚本能够随机生成 500 对长度也随机的消息 m 和密钥 k，然后对 SHA-3 的各种输出长度，把实现的 SHA-3 算法和 SHA-3 HMAC 算法的结果与 Python 中 `hashlib` 和 `hmac` 库的结果相比较。一旦有错误，测试脚本就会报错，打印出出现错

误的消息 m（在测试 HMAC 算法报错时也包括密钥 k）。经过多次运行，没有发生错误，因此可以判断实现一般是正确的。

总结

SHA-1 哈希算法

这个算法是我第一次尝试写哈希算法. 这次写算法让我巩固了 Merkle-Damgård 结构的相关知识, 感觉更加明白了哈希的原理.

HMAC 算法让我更加明白了安全协议和在原语上构建协议的重要性, 它用简单的哈希算法构造出了相对复杂的原语, 但是安全性同样相当高.

哈希函数生日攻击

哈希函数的生日攻击是对密码学函数的又一次攻击. 这次攻击我感觉很有意思, 能够用数学原理把攻击变成实用.

英文消息变形生成算法

这个算法主要是辅助生日攻击用的. 其实这个算法写起来比较有挑战性, 尤其是找映射, 既要考虑到消息本身的结构特征, 又要考虑可实现性. 因此, 这个算法还是比较有挑战性的.

SHA-3 哈希算法

这个算法是现在比较先进的公开算法，实现起来也比较有挑战性。这个算法给我的最大启示，是看文档一定要看好懂的，这样能够快速入门，尤其是概念比较艰深的那些。我看 FIPS 202，把我绕进去了。回过头来看 Keccak 团队的伪代码，很快就实现出来了。还有一个启示，就是涉及细节的时候，一定要实践，根据实践推测事半功倍。我在考虑字节跟纵数组的转换的时候，百思不得其解，后来看到 Keccak 团队的伪代码和网上一个一步一步求

SHA-3 的教程，终于明白转换方式了。这个算法给我最大的经验教训是不光要重视算法，还要重视工具层面。

算法分析与优化

这次对算法的优化也是有相当重要的地位的. 比如对 SHA-1 函数的优化, 可以让它更适合在各种场景应用. 对 HMAC 算法的优化, 是基于对算法的深刻理解.

同时, 对算法优化的经典思路仍然适用. 对英文消息的变形方案, 是基于数学. 对 SHA-3 函数的优化, 是基于看问题观点的转化.

系统设计与维护

这次实验也是需要一定的系统设计的. 在对哈希函数的设计中, 可以把算法的结构和具体的 f 函数分开. 在对 HMAC 算法的设计中, 可以把哈希函数的 f 函数设计成可以改变初始向量 iv, 这样就可以方便优化. 在实现 SHA-3 哈希算法时, 要把层分清楚, 这样不但能够快速实现, 而且能够实现其它 Keccak 团队发明的算法, 因为 f 函数是共通的.

对课程的建议

感觉要求可以更贴近原理, 这样更好一些.

总结

这次哈希算法实验感觉不错, 让我更加明白了哈希算法的原理, 掌握了对哈希算法的初步攻击技巧. 我会努力面对以后的挑战.