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

f 函数

这里 f 函数中使用的加法, 结果都会取最后 32 位, $<<<$ 代表对 32 位整数进行的循环移位运算.

```
def _f(block: bytes, last_result: bytes) -> bytes:
    for i in range(16):
        w[i] = block[4 * i: 4 * i + 4] 按大端序转换成的整数
    for i in range(16, 80):
        w[i] = (w_{i-3} ⊕ w_{i-8} ⊕ w_{i-14} ⊕ w_{i-16}) <<< 1
    for i in range(5):
        h[i] = last_result[4 * i: 4 * i + 4] 按大端序转换成的整数
    a, b, c, d, e = h[0], h[1], h[2], h[3], h[4]
    for i in range(80):
        if 0 ≤ i ≤ 19:
            f = (b ∧ c) ∨ ((¬b) ∧ d)
            k = 0x5a827999
        elif 20 ≤ i ≤ 39:
            f = b ⊕ c ⊕ d:
```

```
            k = 0x6ed9eba1
        elif 40 ≤ i ≤ 59:
            f = (b ∧ c) ∨ (b ∧ d) ∨ (c ∧ d)
            k = 0x8f1bbcdc
        else:
            f = b ⊕ c ⊕ d
            k = 0xca62c1d6
        tmp = (a <<< 5) + f + e + k + w_i
        e = d
        d = c
        c = b <<< 30
        b = a
        a = tmp
    h[0] = h[0] + a
    h[1] = h[1] + b
    h[2] = h[2] + c
    h[3] = h[3] + d
    h[4] = 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成的字节串
```

消息分组函数

```
def _pad_and_iter_chunks(m: bytes) -> Iterable[bytes]:
    result = m
    result 后面填充一个比特位 1
    result 后面不断填充比特位 0, 直到 result 的长度 l 满足 l ≡ 448(mod 512)
    result 后面增加以 64 位大端序整数表示的 l
    return result 按 512 字节分组后形成的若干组
```

Merkle–Damgård 结构哈希算法

```
def merkle_damgard_wide_pipe(iv: bytes, blocks: Iterable[bytes],
                             f: Callable[[bytes, bytes], bytes],
                             block_length: int,
                             output_length: int) -> bytes:
    if iv 的长度与 output_length 不同:
        raise 长度错误
    result = iv
```

```
    for block in blocks:
        if block 的长度与 block_length 不同:
            raise 长度错误
        result = f(block, result)
        if result 的长度与 block_length 不同:
            raise 长度错误
    return result
```

SHA-1 算法

```
def digest(m: bytes) -> bytes:
    return hash_common.merkle_damgard_wide_pipe(iv,
                                                m 生成的各个分组, f,
                                                block_length,
                                                output_length)
```

HMAC 算法

```
def hmac(m: bytes, k: bytes, f_digest: Callable[[bytes], bytes],
         block_length: int) -> bytes:
    if len(k) > block_length:
        k = f(k)
    k = k ‖ b − k 的长度那么多个 0 字节
    k_s1 = k ⊕ ipad
    hash_s1 = f_digest(k_s1 ‖ m)
    k_s2 = k ⊕ opad
    hash_s2 = f_digest(k_s2 ‖ hash_s1)
    return hash_s2
```

SHA-1 HMAC 算法

```
def hmac(m: bytes, k: bytes) -> bytes:
    return hash_common.hmac(m, k, digest, block_length)
```

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 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 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 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 O(1). 但是分组有 O(l) 个, 所以时间复杂度是 O(l). 由于数据之间没有依赖, 而且 f 函数的空间复杂度是 O(1), 所以空间复杂度是 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 O(l) 和 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 O(l_K), 空间复杂度是 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 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 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 O(1).

所以算法总的时空复杂度分别是 O$(l_K + l)$ 和 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 O$(l_K + l)$ 和 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

```
def find_collision_pair(
    rand_bytes_generator_func: Callable[[bytes], bytes],
    hash_func: Callable[[bytes], bytes]
    ) -> Tuple[bytes, bytes]:
    b_1, b_2 = 两对随机消息
    tries = 1
    while H(b_1) != H(b_2) or b_1 == b_2:
        b_1, b_2 = 两对随机消息
        tries += 1
        if tries % 1024 == 0:
            print(tries)
    return b_1, b_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 $O(1)$, 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 $O(2^{\frac{n}{2}})$, 空间复杂度为 $O(1)$.

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.

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.

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.

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.

对课程的建议

感觉以后可以把任务再减少一点, 因为我实在写不完了……写基本要求都要花很大劲去写. 不过现在的要求更加贴近原理, 感觉这样更好一些.

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. 我会努力面对以后的挑战.