Jessup Barrueco

Cryptography White Hat Writeup

SHA-1

SHA stands for Secure Hash Algorithm, specified in August 2015 by the National Institute of Standards and Technology (NIST). SHA-1 is able to claim *security* because it is computationally infeasible to

1) find a message that corresponds to a given message digest
2) find two different messages that produce the same message digest

SHA-1 has two main stages as described by NIST: preprocessing and hash computation. In preprocessing, the message is padded, parsed into *m*-bit blocks, and initialization values are set for the next part, hash computation. In hash computation, a *message schedule* is generated which is used along with functions ROTL(), Ch(), Parity() and Maj() to iteratively generate a series of hash values. SHA-1 has the following limitations:

Message Size (bits):	$< 2^{64}$
Block Size (bits):	512
Word Size (bits):	32
Message Digest Size (bits):	160

The functions integral to the implementation of SHA-1 are described as follows:
ROTL(x) or Rotate Left performs a circular left shift operation, where *x* is a *w*-bit word and *n* is an integer with $0 \le n \le w$

ROTL(x) = ((x << n) | (x >> (w-n)))

The logical functions Ch(), Parity() and Maj() all operate on three 32-bit words *x, y,* and *z* and produce a 32-bit word as output. SHA-1 iterates 80 times per message block. As *t* iterates from 0 to 79:

$Ch(x, y, z) = ((x \& y) \oplus (\neg x \& z))$	when $0 \le t \le 19$
$Parity(x, y, z) = (x \oplus y \oplus z)$	when $20 \le t \le 39$ *and* when $60 \le t \le 79$
$Maj(x, y, z) = ((x \& y) \oplus (x \& z) \oplus (y \& z))$	when $40 \le t \le 59$

SHA-1 also relies on a sequence of 80 constant 32-bit words, $K = \{ K_i : 0 \le i \le 79 \}$ denoted by the following hex values:

5a827999	$0 \le i \le 19$
6ed9eba1	$20 \le i \le 39$
8f1bbcdc	$40 \le i \le 59$
ca62c1d6	$60 \le i \le 79$

SHA-1 Preprocessing has 3 main steps:

1. Padding the Message: the padded message must be a multiple of 512 bits. For an *l*-bit message M, the bit “1” is appended to the end of M, followed by *k*

"0"-bits, where *k* is the smallest, non-negative solution to the equation *l + k +* 1 = 448 mod 512. Then, a 64-bit block equal to the number *l* expressed in binary.

2. Parsing the message into *N* 512-bit blocks, where the length of the padded message is equal to N*512.
3. Setting the Initial Hash Value H[] which will consist of the following five 32-bit words, in hex:
   H[0] = 67452301
   H[1] = efcdab89
   H[2] = 98badcfe
   H[3] = 1032546
   H[4] = c3d2elf0

The SHA-1 Hash Computation uses the functions and constants previously defined. Each message block is processed in order, and the final 160-bit message digest is the concatenation of the result of those operations on each element in H[].

The biggest challenge in our implementation of SHA-1 was computing the correct value of *k*, rather, the number of "0"-bits to append while padding the message in the Preprocessing stage. The line of code in Python3 which finally accomplished this problem for messages of any length is:

k = (((447 - (len(x)*x_bitlength)) % 512) + 512) % 512

Another challenge I experienced in SHA-1 implementation pertained to the technical differences between Java and Python3. All of my homework assignments up until this project were implemented in Java, and I was accustomed to explicit variable type declaration. I experienced difficulty at the beginning, incorrectly assigning the hex values as string variables and then encountering type disagreements in the logical functions described above. In the end, the issue boiled down to using extra print statements for debugging, but I found this challenge unique to my experience switching into a Python3 development environment from Java. However, one of Python3's clear advantages is its command line interface, which allowed me to test my functions quickly with toy variables and get to the root of the issue in a speedier and more efficient manner than I had experienced with Java.

RSA

In order to implement Textbook RSA in Python3, we first had to program Euclid's GCD algorithm and Euclid's Extended Algorithm for Multiplicative Inverses.

> gcd(a,b) takes two integers a, b and returns (g, x, y) such that
    a*x + b*y = gcd(a,b) = g

> multInv(a, b) takes two integers (a, b), uses gcd() to verify coprimality, and returns the multiplicative inverse of a modulo b.

The RSA key generation function keyGen(p, q) requires two primes p, q, so we also needed to implement a function to verify primality, called isPrime(). If both p, q pass this verification step, the public key variable n = p*q is calculated along with Euler's totient function phi(n) = (p-1)(q-1). Our public encryption key *e* is determined by a while loop which randomly generates an integer *e* between 1 and phi(n), checks if gcd(e, phi(n))=1, and if not, restarts the process. Euclid's Extended Algorithm to find Multiplicative Inverses is then used in order to find the decryption private key variable d = multInv(e, phi(n)). The keyGen(p, q) algorithm returns the public encryption key (e, n) and the private decryption key (d, p, q). The ciphertext C is generated and then decrypted with the following equations:

$$C = M^e \quad | \quad M = C^d$$

The biggest challenge in implementing RSA in Python3 was how to handle the incredibly large integers resulting from the encryption and decryption operations. The python3 operations which proved most efficient are as follows:

```
cipher = [pow(ord(char),e,n) for char in plaintext]
plain = [chr(pow(char, d, n)) for char in ciphertext]
```

The function we use in RSA to verify primality of p and q is potentially inefficient for very large inputs, as it performs a brute force check.