Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Departamento de computación

Tópicos selectos de Criptografía

Tarea 4: SHA-1

Tarea

Autor:
André Fabián Castellanos Aldama
Matrícula:
211270011

2022-07-14

Índice

1. Resumen

Se presenta una implementación de la función hash SHA-1. El documento con el algoritmo se encuentra en NIST La implementación se puede encontrar en link.

2. Implementación

```
#include <iostream>
#include <vector>
#include <limits>

uint32_t Ch(const uint32_t &x, const uint32_t &y, const uint32_t &z){
  return (x & y) ^ ((~x) & z);
}

uint32_t Parity(const uint32_t &x, const uint32_t &y, const uint32_t &
    z){
  return x ^ y ^ z;
}

uint32_t Maj(const uint32_t &x, const uint32_t &y, const uint32_t &z){
  return (x & y) ^ (x & z) ^ (y & z);
}

uint32_t ROTR(const uint32_t &x, const uint32_t &n){
  return (x >> n) | ( x << (32-n));
}

uint32_t ft(const uint32_t &x, const uint32_t &y, const uint32_t &z,
    size_t t){

  if(60 <= t && t <= 79){
    return Parity(x, y, z);
  } else if(40 <= t && t <= 59){
    return Maj(x, y, z);
  } else if(20 <= t && t <= 39){
    return Parity(x, y, z);
  } else if(0 <= t && t <= 19){
    return Ch(x, y, z);
  }

  throw std::runtime_error("No existe la funci n para t");

}

```

```cpp
uint32_t ROTL(const uint32_t &x, const size_t &n){
  return (x << n) | ( x >> (32-n));
}

// circular bit shift left of integer 'number' by 'n' bit positions
template<typename T>
T circular_shift_left(T number, std::size_t n) {
  static_assert(std::is_integral<T>::value, "an integral type is required");

  // the corresponding unsigned type if T is a signed integer, T itself otherwise
  using unsigned_type = std::make_unsigned_t<T>;

  // number of bits in the integral type
  constexpr std::size_t NBITS = std::numeric_limits<unsigned_type>::digits;

  n %= NBITS; // bring the number of bit positions to shift by to within a valid range
  const unsigned_type un = number; // the number interpreted as an unsigned value

  // circular bit shift left of an unsigned NBITS-bit integer by n bit positions
  return (un << n) | (un >> (NBITS - n));
}

int main() {

  // Arreglo de prueba
//  std::vector<uint8_t> Message {'a','b','c'};
//  std::vector<uint8_t> Message {'a','b','c','d','b','c','d','e','c','d','e','f','d','e','f','g','e','f','g','h','f','g',
//                                'h','i','g','h','i','j','h','i','j','k','i','j','k','l','j','k','l','m','k','l','m','n',
//                                'l','m','n','o','m','n','o','p','n','o','p','q'};
//  std::vector<uint8_t> Message(1000000,'a');

  /* ------------------------------ Padding ------------------------------ */
  // N mero de 64 elementos completos en el arreglo Message
  size_t incomplete8BitBlockSize = Message.size() % 64;

  // N mero de bits en el mensaje
  unsigned long long numberOfBits = Message.size() * 8;

  // Convertir n mero de bits en dos de 32: numberOfBitsUpper || numberOfBitsLow
  uint32_t numberOfBitsUpper = numberOfBits >> 32;
  uint32_t numberOfBitsLow = (numberOfBits << 32) >> 32;
  uint8_t numberOfBitsPart1 = numberOfBitsUpper >> 24;
  uint8_t numberOfBitsPart2 = (numberOfBitsUpper << 8) >> 24;
  uint8_t numberOfBitsPart3 = (numberOfBitsUpper << 16) >> 24;
  uint8_t numberOfBitsPart4 = (numberOfBitsUpper << 24) >> 24;
  uint8_t numberOfBitsPart5 = numberOfBitsLow >> 24;
  uint8_t numberOfBitsPart6 = (numberOfBitsLow << 8) >> 24;
  uint8_t numberOfBitsPart7 = (numberOfBitsLow << 16) >> 24;
  uint8_t numberOfBitsPart8 = (numberOfBitsLow << 24) >> 24;

```

```
//  448 bits o 56 elementos de 8 bits >=
if(incomplete8BitBlockSize >= 56) {
  // Add 1000...000 (8 bits)
  Message.push_back(128);

  // Add 000.. (8 bits) until 0 bits remaining to 512
  for(size_t i = incomplete8BitBlockSize + 1; i < 64; ++i){
    Message.push_back(0);
  }

  // Add 000.. (8 bits) until 64 bits remaining to 512
  for(size_t i = 0; i < 56; ++i){
    Message.push_back(0);
  }

  // Add length
  Message.push_back(numberOfBitsPart1);
  Message.push_back(numberOfBitsPart2);
  Message.push_back(numberOfBitsPart3);
  Message.push_back(numberOfBitsPart4);
  Message.push_back(numberOfBitsPart5);
  Message.push_back(numberOfBitsPart6);
  Message.push_back(numberOfBitsPart7);
  Message.push_back(numberOfBitsPart8);

} else if ( 0 <= incomplete8BitBlockSize && incomplete8BitBlockSize
 < 56) {
  // Add 1000...000 (8 bits)
  Message.push_back(128);

  // Add 000.. (8 bits) until 64 bits remaining
  for(size_t i = incomplete8BitBlockSize + 1; i < 56; ++i){
    Message.push_back(0);
  }

  // Add length
  Message.push_back(numberOfBitsPart1);
  Message.push_back(numberOfBitsPart2);
  Message.push_back(numberOfBitsPart3);
  Message.push_back(numberOfBitsPart4);
  Message.push_back(numberOfBitsPart5);
  Message.push_back(numberOfBitsPart6);
  Message.push_back(numberOfBitsPart7);
  Message.push_back(numberOfBitsPart8);

}

/* ------------------------------ Constants and Initial hash value
 ------------------------------ */

std::vector<uint32_t> K0(20, 0x5a827999);
std::vector<uint32_t> K1(20, 0x6ed9eba1);
std::vector<uint32_t> K2(20, 0x8f1bbcdc);
std::vector<uint32_t> K3(20, 0xca62c1d6);

std::vector<uint32_t> K;
K.insert(K.end(), K0.begin(), K0.end());
K.insert(K.end(), K1.begin(), K1.end());
K.insert(K.end(), K2.begin(), K2.end());
K.insert(K.end(), K3.begin(), K3.end());

```

```
146   std::vector<uint32_t> H {0x67452301, 0xefcdab89, 0x98badcfe, 0
       x10325476, 0xc3d2e1f0};
147
148   // Parse to 512 bit messsage block
149   for(size_t indexMessage512Bits = 0; indexMessage512Bits < Message.
       size(); indexMessage512Bits += 64){
150     uint32_t M1, M2, M3, M4, M5, M6, M7, M8, M9, M10, M11, M12, M13,
       M14, M15, M16;
151
152     /* Initialize M's*/
153     M1 = Message[indexMessage512Bits];
154     M1 = (M1 << 8) + Message[indexMessage512Bits + 1];
155     M1 = (M1 << 8) + Message[indexMessage512Bits + 2];
156     M1 = (M1 << 8) + Message[indexMessage512Bits + 3];
157
158     M2 = Message[indexMessage512Bits + 4];
159     M2 = (M2 << 8) + Message[indexMessage512Bits + 5];
160     M2 = (M2 << 8) + Message[indexMessage512Bits + 6];
161     M2 = (M2 << 8) + Message[indexMessage512Bits + 7];
162
163     M3 = Message[indexMessage512Bits + 8];
164     M3 = (M3 << 8) + Message[indexMessage512Bits + 9];
165     M3 = (M3 << 8) + Message[indexMessage512Bits + 10];
166     M3 = (M3 << 8) + Message[indexMessage512Bits + 11];
167
168     M4 = Message[indexMessage512Bits + 12];
169     M4 = (M4 << 8) + Message[indexMessage512Bits + 13];
170     M4 = (M4 << 8) + Message[indexMessage512Bits + 14];
171     M4 = (M4 << 8) + Message[indexMessage512Bits + 15];
172
173     M5 = Message[indexMessage512Bits + 16];
174     M5 = (M5 << 8) + Message[indexMessage512Bits + 17];
175     M5 = (M5 << 8) + Message[indexMessage512Bits + 18];
176     M5 = (M5 << 8) + Message[indexMessage512Bits + 19];
177
178     M6 = Message[indexMessage512Bits + 20];
179     M6 = (M6 << 8) + Message[indexMessage512Bits + 21];
180     M6 = (M6 << 8) + Message[indexMessage512Bits + 22];
181     M6 = (M6 << 8) + Message[indexMessage512Bits + 23];
182
183     M7 = Message[indexMessage512Bits + 24];
184     M7 = (M7 << 8) + Message[indexMessage512Bits + 25];
185     M7 = (M7 << 8) + Message[indexMessage512Bits + 26];
186     M7 = (M7 << 8) + Message[indexMessage512Bits + 27];
187
188     M8 = Message[indexMessage512Bits + 28];
189     M8 = (M8 << 8) + Message[indexMessage512Bits + 29];
190     M8 = (M8 << 8) + Message[indexMessage512Bits + 30];
191     M8 = (M8 << 8) + Message[indexMessage512Bits + 31];
192
193     M9 = Message[indexMessage512Bits + 32];
194     M9 = (M9 << 8) + Message[indexMessage512Bits + 33];
195     M9 = (M9 << 8) + Message[indexMessage512Bits + 34];
196     M9 = (M9 << 8) + Message[indexMessage512Bits + 35];
197
198     M10 = Message[indexMessage512Bits + 36];
199     M10 = (M10 << 8) + Message[indexMessage512Bits + 37];
200     M10 = (M10 << 8) + Message[indexMessage512Bits + 38];
201     M10 = (M10 << 8) + Message[indexMessage512Bits + 39];
202
203     M11 = Message[indexMessage512Bits + 40];
```

```
    M11 = (M11 << 8) + Message[indexMessage512Bits + 41];
    M11 = (M11 << 8) + Message[indexMessage512Bits + 42];
    M11 = (M11 << 8) + Message[indexMessage512Bits + 43];

    M12 = Message[indexMessage512Bits + 44];
    M12 = (M12 << 8) + Message[indexMessage512Bits + 45];
    M12 = (M12 << 8) + Message[indexMessage512Bits + 46];
    M12 = (M12 << 8) + Message[indexMessage512Bits + 47];

    M13 = Message[indexMessage512Bits + 48];
    M13 = (M13 << 8) + Message[indexMessage512Bits + 49];
    M13 = (M13 << 8) + Message[indexMessage512Bits + 50];
    M13 = (M13 << 8) + Message[indexMessage512Bits + 51];

    M14 = Message[indexMessage512Bits + 52];
    M14 = (M14 << 8) + Message[indexMessage512Bits + 53];
    M14 = (M14 << 8) + Message[indexMessage512Bits + 54];
    M14 = (M14 << 8) + Message[indexMessage512Bits + 55];

    M15 = Message[indexMessage512Bits + 56];
    M15 = (M15 << 8) + Message[indexMessage512Bits + 57];
    M15 = (M15 << 8) + Message[indexMessage512Bits + 58];
    M15 = (M15 << 8) + Message[indexMessage512Bits + 59];

    M16 = Message[indexMessage512Bits + 60];
    M16 = (M16 << 8) + Message[indexMessage512Bits + 61];
    M16 = (M16 << 8) + Message[indexMessage512Bits + 62];
    M16 = (M16 << 8) + Message[indexMessage512Bits + 63];

    /* Prepare the message schedule */
    std::vector<uint32_t> W(80);
    W[0] = M1;
    W[1] = M2;
    W[2] = M3;
    W[3] = M4;
    W[4] = M5;
    W[5] = M6;
    W[6] = M7;
    W[7] = M8;
    W[8] = M9;
    W[9] = M10;
    W[10] = M11;
    W[11] = M12;
    W[12] = M13;
    W[13] = M14;
    W[14] = M15;
    W[15] = M16;

    for(size_t t = 16; t < 80; ++t){
      uint32_t tmp = W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16];
      W[t] = circular_shift_left(tmp, 1);
    }

    std::vector<uint32_t> tmp(H);

    for(size_t t = 0; t < 80; ++t){
      uint64_t T = circular_shift_left(tmp[0], 5) + ft(tmp[1], tmp[2],
    tmp[3], t) + tmp[4] + K[t] + W[t];
      tmp[4] = tmp[3];
      tmp[3] = tmp[2];
      tmp[2] = circular_shift_left(tmp[1], 30);
```

```
      tmp[1] = tmp[0];
      tmp[0] = T;
    }

    H[0] = tmp[0] + H[0];
    H[1] = tmp[1] + H[1];
    H[2] = tmp[2] + H[2];
    H[3] = tmp[3] + H[3];
    H[4] = tmp[4] + H[4];

  }

  for(size_t i = 0; i < H.size(); ++i){
    std::cout << std::hex << H[i] << " \n"[i == (H.size() - 1)];
  }

  return 0;
}
```

Listing 1: Implementación.

Puede seleccionar cada arreglo de las 3 pruebas del libro

3. Pruebas

El libro proporciona 3 hash para 3 arreglos:

1. abc: `a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d`.
2. abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq: `84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1`.
3. aaaa... (1000000): `34aa973c d4c4daa4 f61eeb2b dbad2731 6534016f`.

```
Problema 1:
/home/andre/Documents/2022/Cursos-May-Sep/Criptografia/SHA1/cmake-build-debug/SHA1
a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d

Problema 2:
/home/andre/Documents/2022/Cursos-May-Sep/Criptografia/SHA1/cmake-build-debug/SHA1
84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1

Problema 3:
/home/andre/Documents/2022/Cursos-May-Sep/Criptografia/SHA1/cmake-build-debug/SHA1
34aa973c d4c4daa4 f61eeb2b dbad2731 6534016f
```