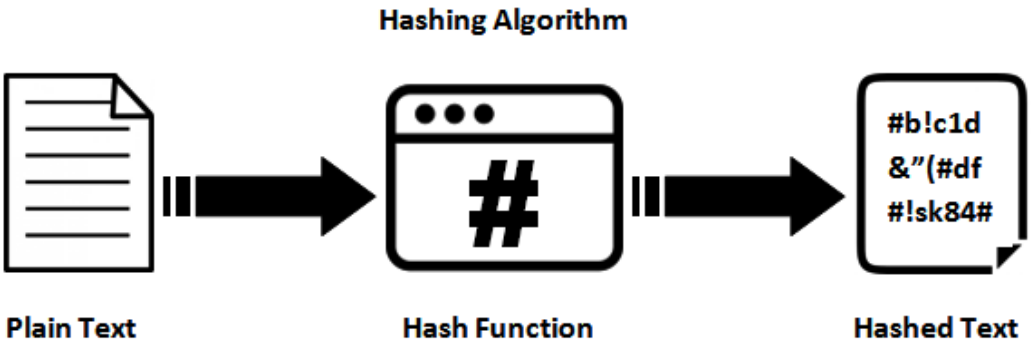
Experiment - 9

Aim

To implement a program to generate SHA-1 hash

Description

The Secure Hash Algorithm 1 (SHA-1) is a cryptographic computer security algorithm. It was created by the US National Security Agency in 1995. SHA-1 produces a 160-bit hash value or message digests from the inputted data. It is widely used in security applications and protocols, including TLS, SSL, PGP, SSH, IPsec, and S/MIME.



Code

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <fstream>

using namespace std;

/* Help macros */
#define SHA1_ROL(value, bits) (((value) << (bits)) | (((value)&0xffffffff) >> (32 - (bits))))
#define SHA1_BLK(i) (block[i & 15] = SHA1_ROL(block[(i + 13) & 15] ^ block[(i + 8) & 15] ^ b
lock[(i + 2) & 15] ^ block[i & 15], 1))

/* (R0+R1), R2, R3, R4 are the different operations used in SHA1 */
#define SHA1_R0(v, w, x, y, z, i)                                \
    z += ((w & (x ^ y)) ^ y) + block[i] + 0x5a827999 + SHA1_ROL(v, 5); \
    w = SHA1_ROL(w, 30);
#define SHA1_R1(v, w, x, y, z, i)                                  \
    z += ((w & (x ^ y)) ^ y) + SHA1_BLK(i) + 0x5a827999 + SHA1_ROL(v, 5); \
    w = SHA1_ROL(w, 30);
#define SHA1_R2(v, w, x, y, z, i)                              \
```

```
    z += (w ^ x ^ y) + SHA1_BLK(i) + 0x6ed9eba1 + SHA1_ROL(v, 5); \
    w = SHA1_ROL(w, 30);
#define SHA1_R3(v, w, x, y, z, i)                                   \
    z += (((w | x) & y) | (w & x)) + SHA1_BLK(i) + 0x8f1bbcdc + SHA1_ROL(v, 5); \
    w = SHA1_ROL(w, 30);
#define SHA1_R4(v, w, x, y, z, i)                                   \
    z += (w ^ x ^ y) + SHA1_BLK(i) + 0xca62c1d6 + SHA1_ROL(v, 5); \
    w = SHA1_ROL(w, 30);

class SHA1
{
public:
    SHA1();
    void update(const string &s);
    void update(istream &is);
    string final();
    static string from_file(const string &filename);

private:
    typedef unsigned long int uint32;  /* just needs to be at least 32bit */
    typedef unsigned long long uint64; /* just needs to be at least 64bit */

    static const unsigned int DIGEST_INTS = 5; /* number of 32bit integers per SHA1 digest */
    static const unsigned int BLOCK_INTS = 16; /* number of 32bit integers per SHA1 block */
    static const unsigned int BLOCK_BYTES = BLOCK_INTS * 4;

    uint32 digest[DIGEST_INTS];
    string buffer;
    uint64 transforms;

    void reset();
    void transform(uint32 block[BLOCK_BYTES]);

    static void buffer_to_block(const string &buffer, uint32 block[BLOCK_BYTES]);
    static void read(istream &is, string &s, int max);
};

string sha1(const string &string);

SHA1::SHA1()
{
    reset();
}
```

```
void SHA1::update(const string &s)
{
   istringstream is(s);
   update(is);
}

void SHA1::update(istream &is)
{
   string rest_of_buffer;
   read(is, rest_of_buffer, BLOCK_BYTES - buffer.size());
   buffer += rest_of_buffer;

   while (is)
   {
      uint32 block[BLOCK_INTS];
      buffer_to_block(buffer, block);
      transform(block);
      read(is, buffer, BLOCK_BYTES);
   }
}

/*
 * Add padding and return the message digest.
 */

string SHA1::final()
{
   /* Total number of hashed bits */
   uint64 total_bits = (transforms * BLOCK_BYTES + buffer.size()) * 8;

   /* Padding */
   buffer += 0x80;
   unsigned int orig_size = buffer.size();
   while (buffer.size() < BLOCK_BYTES)
   {
      buffer += (char)0x00;
   }

   uint32 block[BLOCK_INTS];
   buffer_to_block(buffer, block);

   if (orig_size > BLOCK_BYTES - 8)
```

```
    {
        transform(block);
        for (unsigned int i = 0; i < BLOCK_INTS - 2; i++)
        {
            block[i] = 0;
        }
    }

    /* Append total_bits, split this uint64 into two uint32 */
    block[BLOCK_INTS - 1] = total_bits;
    block[BLOCK_INTS - 2] = (total_bits >> 32);
    transform(block);

    /* Hex string */
    ostringstream result;
    for (unsigned int i = 0; i < DIGEST_INTS; i++)
    {
        result << hex << setfill('0') << setw(8);
        result << (digest[i] & 0xffffffff);
    }

    /* Reset for next run */
    reset();

    return result.str();
}

string SHA1::from_file(const string &filename)
{
    ifstream stream(filename.c_str(), ios::binary);
    SHA1 checksum;
    checksum.update(stream);
    return checksum.final();
}

void SHA1::reset()
{
    /* SHA1 initialization constants */
    digest[0] = 0x67452301;
    digest[1] = 0xefcdab89;
    digest[2] = 0x98badcfe;
    digest[3] = 0x10325476;
    digest[4] = 0xc3d2e1f0;
```

```
    /* Reset counters */
    transforms = 0;
    buffer = "";
}
/*
 * Hash a single 512-bit block. This is the core of the algorithm.
 */
void SHA1::transform(uint32 block[BLOCK_BYTES])
{
    /* Copy digest[] to working vars */
    uint32 a = digest[0];
    uint32 b = digest[1];
    uint32 c = digest[2];
    uint32 d = digest[3];
    uint32 e = digest[4];

    /* 4 rounds of 20 operations each. Loop unrolled. */
    SHA1_R0(a, b, c, d, e, 0);
    SHA1_R0(e, a, b, c, d, 1);
    SHA1_R0(d, e, a, b, c, 2);
    SHA1_R0(c, d, e, a, b, 3);
    SHA1_R0(b, c, d, e, a, 4);
    SHA1_R0(a, b, c, d, e, 5);
    SHA1_R0(e, a, b, c, d, 6);
    SHA1_R0(d, e, a, b, c, 7);
    SHA1_R0(c, d, e, a, b, 8);
    SHA1_R0(b, c, d, e, a, 9);
    SHA1_R0(a, b, c, d, e, 10);
    SHA1_R0(e, a, b, c, d, 11);
    SHA1_R0(d, e, a, b, c, 12);
    SHA1_R0(c, d, e, a, b, 13);
    SHA1_R0(b, c, d, e, a, 14);
    SHA1_R0(a, b, c, d, e, 15);
    SHA1_R1(e, a, b, c, d, 16);
    SHA1_R1(d, e, a, b, c, 17);
    SHA1_R1(c, d, e, a, b, 18);
    SHA1_R1(b, c, d, e, a, 19);
    SHA1_R2(a, b, c, d, e, 20);
    SHA1_R2(e, a, b, c, d, 21);
    SHA1_R2(d, e, a, b, c, 22);
    SHA1_R2(c, d, e, a, b, 23);
    SHA1_R2(b, c, d, e, a, 24);
```

```
SHA1_R2(a, b, c, d, e, 25);
SHA1_R2(e, a, b, c, d, 26);
SHA1_R2(d, e, a, b, c, 27);
SHA1_R2(c, d, e, a, b, 28);
SHA1_R2(b, c, d, e, a, 29);
SHA1_R2(a, b, c, d, e, 30);
SHA1_R2(e, a, b, c, d, 31);
SHA1_R2(d, e, a, b, c, 32);
SHA1_R2(c, d, e, a, b, 33);
SHA1_R2(b, c, d, e, a, 34);
SHA1_R2(a, b, c, d, e, 35);
SHA1_R2(e, a, b, c, d, 36);
SHA1_R2(d, e, a, b, c, 37);
SHA1_R2(c, d, e, a, b, 38);
SHA1_R2(b, c, d, e, a, 39);
SHA1_R3(a, b, c, d, e, 40);
SHA1_R3(e, a, b, c, d, 41);
SHA1_R3(d, e, a, b, c, 42);
SHA1_R3(c, d, e, a, b, 43);
SHA1_R3(b, c, d, e, a, 44);
SHA1_R3(a, b, c, d, e, 45);
SHA1_R3(e, a, b, c, d, 46);
SHA1_R3(d, e, a, b, c, 47);
SHA1_R3(c, d, e, a, b, 48);
SHA1_R3(b, c, d, e, a, 49);
SHA1_R3(a, b, c, d, e, 50);
SHA1_R3(e, a, b, c, d, 51);
SHA1_R3(d, e, a, b, c, 52);
SHA1_R3(c, d, e, a, b, 53);
SHA1_R3(b, c, d, e, a, 54);
SHA1_R3(a, b, c, d, e, 55);
SHA1_R3(e, a, b, c, d, 56);
SHA1_R3(d, e, a, b, c, 57);
SHA1_R3(c, d, e, a, b, 58);
SHA1_R3(b, c, d, e, a, 59);
SHA1_R4(a, b, c, d, e, 60);
SHA1_R4(e, a, b, c, d, 61);
SHA1_R4(d, e, a, b, c, 62);
SHA1_R4(c, d, e, a, b, 63);
SHA1_R4(b, c, d, e, a, 64);
SHA1_R4(a, b, c, d, e, 65);
SHA1_R4(e, a, b, c, d, 66);
SHA1_R4(d, e, a, b, c, 67);
```

```
    SHA1_R4(c, d, e, a, b, 68);
    SHA1_R4(b, c, d, e, a, 69);
    SHA1_R4(a, b, c, d, e, 70);
    SHA1_R4(e, a, b, c, d, 71);
    SHA1_R4(d, e, a, b, c, 72);
    SHA1_R4(c, d, e, a, b, 73);
    SHA1_R4(b, c, d, e, a, 74);
    SHA1_R4(a, b, c, d, e, 75);
    SHA1_R4(e, a, b, c, d, 76);
    SHA1_R4(d, e, a, b, c, 77);
    SHA1_R4(c, d, e, a, b, 78);
    SHA1_R4(b, c, d, e, a, 79);

    /* Add the working vars back into digest[] */
    digest[0] += a;
    digest[1] += b;
    digest[2] += c;
    digest[3] += d;
    digest[4] += e;

    /* Count the number of transformations */
    transforms++;
}

void SHA1::buffer_to_block(const string &buffer, uint32 block[BLOCK_BYTES])
{
    /* Convert the string (byte buffer) to a uint32 array (MSB) */
    for (unsigned int i = 0; i < BLOCK_INTS; i++)
    {
                    block[i] = (buffer[4 * i + 3] & 0xff) | (buffer[4 * i + 2] & 0xff) << 8 |
 (buffer[4 * i + 1] & 0xff) << 16 | (buffer[4 * i + 0] & 0xff) << 24;
    }
}

void SHA1::read(istream &is, string &s, int max)
{
    char sbuf[max];
    is.read(sbuf, max);
    s.assign(sbuf, is.gcount());
}

string sha1(const string &string)
{
```

```
    SHA1 sha1;
    sha1.update(string);
    return sha1.final();
}

int main()
{
    string message;
    cout << "Enter message : ";
    getline(cin, message);

    cout << "\nMessage : " << message << endl;

    cout << "Hash : " << sha1(message);
    return 0;
}
```

Output

```
lab $ ./a.out
Enter message : Hello World

Message : Hello World
Hash : 0a4d55a8d778e5022fab701977c5d840bbc486d0
```

Discussion

Albeit SHA-1 is still widely used, cryptanalysts in 2005 were able to find vulnerabilities on this algorithm that detrimentally compromised its security. These vulnerabilities came in the form of an algorithm that speedily finds collisions with different inputs, meaning that two distinct inputs map to the same digest. SHA-1 and SHA-2 are the hash algorithms required by law for use in certain U.S. government applications, including use within other cryptographic algorithms and protocols, for the protection of sensitive unclassified information.

Finding and Learnings

In this experiment we learnt how to use SHA-1 algorithm to generate hash of a given message. A prime motivation for the publication of the Secure Hash Algorithm was the Digital Signature Standard, in which it is incorporated. Due to the exposed vulnerabilities of SHA-1, cryptographers modified the algorithm to produce SHA-2, which consists of not one but two hash functions known as SHA-256 and SHA-512, using 32- and 64-bit words, respectively. There are additional truncated versions of these hash functions, known as SHA-224, SHA-384, SHA-512/224, and SHA-512/256, which can be used for either part of the algorithm.

Experiment - 10

Aim

To implement a digital signature algorithm

Description

A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A digital signature is an authentication mechanism that enables the creator of the message to attach a code that acts as a signature.

A digital signature algorithm typically consists of 3 procedures :-

- A key generation algorithm that selects a private key uniformly at random from a set of possible private keys. The algorithm outputs the private key and a corresponding public key.
- A signing algorithm that, given a message and a private key, produces a signature.
- A signature verifying algorithm that, given the message, public key and signature, either accepts or rejects the message's claim to authenticity.

Algorithm

1. Message digest is computed by applying hash function on input message.
2. Message digest is encrypted using private key of sender to form digital signature.
3. Digital signature is then transmitted along with the message.
4. Receiver decrypts the digital signature using the public key of sender.(assures authenticity)
5. The receiver now has the original message digest.
6. The receiver can compute the message digest from the message sent along original message digest.
7. The message digest computed by receiver and the message digest (got by decryption on digital signature) need to be same for ensuring integrity.

Code

```
#include <stdio.h>
#include <stdlib.h>

// 64-bit integer type
typedef long long int dlong;
// rational ec point
typedef struct
{
   dlong x, y;
} epnt;
// elliptic curve parameters
typedef struct
{
```

```
    long a, b;
    dlong N;
    epnt G;
    dlong r;
} curve;
// signature pair
typedef struct
{
    long a, b;
} pair;

// maximum modulus
const long mxN = 1073741789;
// max order G = mxN + 65536
const long mxr = 1073807325;
// symbolic infinity
const long inf = -2147483647;

// single global curve
curve e;
// point at infinity zerO
epnt zerO;
// impossible inverse mod N
int inverr;

// return mod(v^-1, u)
long exgcd(long v, long u)
{
    register long q, t;
    long r = 0, s = 1;
    if (v < 0)
        v += u;

    while (v)
    {
        q = u / v;
        t = u - q * v;
        u = v;
        v = t;
        t = r - q * s;
        r = s;
        s = t;
    }
```

```
    if (u != 1)
    {
        printf(" impossible inverse mod N, gcd = %d\n", u);
        inverr = 1;
    }
    return r;
}

// return mod(a, N)
static inline dlong modn(dlong a)
{
    a %= e.N;
    if (a < 0)
        a += e.N;
    return a;
}

// return mod(a, r)
dlong modr(dlong a)
{
    a %= e.r;
    if (a < 0)
        a += e.r;
    return a;
}

// return the discriminant of E
long disc(void)
{
    dlong c, a = e.a, b = e.b;
    c = 4 * modn(a * modn(a * a));
    return modn(-16 * (c + 27 * modn(b * b)));
}

// return 1 if P = zerO
int isO(epnt p)
{
    return (p.x == inf) && (p.y == 0);
}

// return 1 if P is on curve E
int ison(epnt p)
{
```

```
    long r, s;
    if (!isO(p))
    {
        r = modn(e.b + p.x * modn(e.a + p.x * p.x));
        s = modn(p.y * p.y);
    }
    return (r == s);
}

// full ec point addition
void padd(epnt *r, epnt p, epnt q)
{
    dlong la, t;

    if (isO(p))
    {
        *r = q;
        return;
    }
    if (isO(q))
    {
        *r = p;
        return;
    }

    if (p.x != q.x)
    { // R:= P + Q
        t = p.y - q.y;
        la = modn(t * exgcd(p.x - q.x, e.N));
    }
    else // P = Q, R := 2P
        if ((p.y == q.y) && (p.y != 0))
    {
        t = modn(3 * modn(p.x * p.x) + e.a);
        la = modn(t * exgcd(2 * p.y, e.N));
    }
    else
    {
        *r = zerO;
        return;
    } // P = -Q, R := O

    t = modn(la * la - p.x - q.x);
```

```
    r->y = modn(la * (p.x - t) - p.y);
    r->x = t;
    if (inverr)
        *r = zerO;
}

// R:= multiple kP
void pmul(epnt *r, epnt p, long k)
{
    epnt s = zerO, q = p;

    for (; k; k >>= 1)
    {
        if (k & 1)
            padd(&s, s, q);
        if (inverr)
        {
            s = zerO;
            break;
        }
        padd(&q, q, q);
    }
    *r = s;
}

// print point P with prefix f
void pprint(char *f, epnt p)
{
    dlong y = p.y;

    if (isO(p))
        printf("%s (0)\n", f);

    else
    {
        if (y > e.N - y)
            y -= e.N;
        printf("%s (%lld, %lld)\n", f, p.x, y);
    }
}

// initialize elliptic curve
int ellinit(long i[])
```

```
{
   long a = i[0], b = i[1];
   e.N = i[2];
   inverr = 0;

   if ((e.N < 5) || (e.N > mxN))
      return 0;

   e.a = modn(a);
   e.b = modn(b);
   e.G.x = modn(i[3]);
   e.G.y = modn(i[4]);
   e.r = i[5];

   if ((e.r < 5) || (e.r > mxr))
      return 0;

   printf("\nE: y^2 = x^3 + %dx + %d", a, b);
   printf(" (mod %lld)\n", e.N);
   pprint("base point G", e.G);
   printf("order(G, E) = %lld\n", e.r);

   return 1;
}

// pseudorandom number [0..1)
double rnd(void)
{
   return rand() / ((double)RAND_MAX + 1);
}

// signature primitive
pair signature(dlong s, long f)
{
   long c, d, u, u1;
   pair sg;
   epnt V;

   printf("\nsignature computation\n");
   do
   {
      do
      {
```

```
            u = 1 + (long)(rnd() * (e.r - 1));
            pmul(&V, e.G, u);
            c = modr(V.x);
        } while (c == 0);

        u1 = exgcd(u, e.r);
        d = modr(u1 * (f + modr(s * c)));
    } while (d == 0);
    printf("one-time u = %d\n", u);
    pprint("V = uG", V);

    sg.a = c;
    sg.b = d;
    return sg;
}

// verification primitive
int verify(epnt W, long f, pair sg)
{
    long c = sg.a, d = sg.b;
    long t, c1, h1, h2;
    dlong h;
    epnt V, V2;

    // domain check
    t = (c > 0) && (c < e.r);
    t &= (d > 0) && (d < e.r);
    if (!t)
        return 0;

    printf("\nsignature verification\n");
    h = exgcd(d, e.r);
    h1 = modr(f * h);
    h2 = modr(c * h);
    printf("h1,h2 = %d, %d\n", h1, h2);
    pmul(&V, e.G, h1);
    pmul(&V2, W, h2);
    pprint("h1G", V);
    pprint("h2W", V2);
    padd(&V, V, V2);
    pprint("+ =", V);
    if (isO(V))
        return 0;
```

```
    c1 = modr(V.x);
    printf("c' = %d\n", c1);

    return (c1 == c);
}

// digital signature on message hash f, error bit d
void ec_dsa(long f, long d)
{
    long i, s, t;
    pair sg;
    epnt W;

    // parameter check
    t = (disc() == 0);
    t |= isO(e.G);
    pmul(&W, e.G, e.r);
    t |= !isO(W);
    t |= !ison(e.G);
    if (t)
        goto errmsg;

    printf("\nkey generation\n");
    s = 1 + (long)(rnd() * (e.r - 1));
    pmul(&W, e.G, s);
    printf("private key s = %d\n", s);
    pprint("public key W = sG", W);

    // next highest power of 2 - 1
    t = e.r;
    for (i = 1; i < 32; i <<= 1)
        t |= t >> i;
    while (f > t)
        f >>= 1;
    printf("\naligned hash %x\n", f);

    sg = signature(s, f);
    if (inverr)
        goto errmsg;
    printf("signature c,d = %d, %d\n", sg.a, sg.b);

    if (d > 0)
    {
```

```
        while (d > t)
            d >>= 1;
        f ^= d;
        printf("\ncorrupted hash %x\n", f);
    }

    t = verify(W, f, sg);
    if (inverr)
        goto errmsg;
    if (t)
        printf("Valid\n_____\n");
    else
        printf("invalid\n_______\n");

    return;

errmsg:
    printf("invalid parameter set\n");
    printf("_____\n");
}

int main()
{
    typedef long eparm[6];
    long d, f;
    zerO.x = inf;
    zerO.y = 0;

    // Test vectors: elliptic curve domain parameters,
    // short Weierstrass model y^2 = x^3 + ax + b (mod N)
    eparm *sp, sets[10] = {
                //    a,   b,  modulus N, base point G, order(G, E), cofactor
                {3, 2, 5, 2, 1, 5},
            };
    // Digital signature on message hash f,
    // set d > 0 to simulate corrupted data
    f = 0x789abcde;
    d = 0;

    for (sp = sets;; sp++)
    {
        if (ellinit(*sp))
            ec_dsa(f, d);
```

```
        else
            break;
    }
    return 0;
}
```

Output

```
lab $ ./a.out

E: y^2 = x^3 + 3x + 2 (mod 5)
base point G (2, 1)
order(G, E) = 5

key generation
private key s = 1
public key W = sG (2, 1)

aligned hash 7

signature computation
one-time u = 1
V = uG (2, 1)
signature c,d = 2, 4

signature verification
h1,h2 = 3, 3
h1G (1, 1)
h2W (1, 1)
+ = (2, 1)
c' = 2
Valid
```

Discussion

Digital signatures employ asymmetric cryptography and provide a layer of validation and security to messages sent through a non-secure channel. In digital signature algorithms, message digest is

computed using one-way hash function, i.e. a hash function in which computation of hash value of a message is easy but computation of the message from hash value of the message is very difficult. Digital signatures are equivalent to traditional handwritten signatures in many respects, but properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim they did not sign a message, while also claiming their private key remains secret.

Finding and Learnings

In this experiment we learnt how to apply digital signature algorithm on message to ensure authenticity of a message.

For digital signature, two main properties required are :-

1. The authenticity of a signature generated from a fixed message and fixed private key can be verified by using the corresponding public key.
2. It should be computationally infeasible to generate a valid signature for a party without knowing that party's private key.