APPENDIX A

Applications of Hash Function

(i) Digital Signatures
Digital signatures [4] are very important in information security. A digital signature authenticates electronic documents in a similar manner a handwritten signature authenticates printed documents. Digital signatures enable the authentication and non-repudiation of digital messages, assuring the recipient of a digital message of both the identity of the sender and the integrity of the message. Signature generation makes use of a private key to generate a digital signature. Signature verification makes use of a public key, which corresponds to, but is not the same as, the private key.
Hash functions are used in conjunction with digital signature schemes, where a message is hashed first, and then the hash value, as a representative of the message, is signed in place of the original message. The digital signature is sent to the intended verifier along with the message. The verifier computes the hash value over the received message and verifies the signature by using the sender's public key. Anyone can verify the correctness of the digital signature of a message using the publicly listed verification key and thereby obtains proof of authenticity. However, only someone who knows the signing key can actually generate signatures. The security of digital signatures depends on the cryptographic strength of the underlying hash functions.

(ii) MAC
Message integrity and authenticity are essential in security-related communications. Here, a recipient is expected to be able to verify that a received message, originally transmitted by a valid source, was not changed. Technically, verifying message integrity and authenticity is based on the recipient’s ability to prove to itself that the sender stores a valid secret key that was used when the message was transmitted. MAC is closely related to cryptographic hash functions, which play a fundamental role in many areas of modern cryptography. In software, these hash functions have throughput as much as one order of magnitude higher than DES. Several factors motivated their adoption as the basis for MAC algorithms: the additional implementation and deployment effort required to adopt these as MAC is minimal; MAC based on these outperform
most other available options; and such MAC, avoiding the use of encryption algorithms, may have preferential export status. Consequently, MAC constructions based on these hash functions were adopted in Kerberos, SNMP, and SSL, and gained favor in the IPsec working group of the IETF. \( MAC(M, K) \) is a oneway transformation of the message \( M \) and a secret key \( K \) shared with the verifier. The values \( M \) and \( MAC(M, K) \) are both sent to the verifier. Upon receiving these values, the verifier generates himself a value \( MAC'(M, K) \) based on the received \( M \) and the value of \( K \) known to him. If \( MAC(M, K) = MAC'(M, K) \), the verifier decides that the message is authentic and equals its original value. MAC differs from digital signatures as MAC values are both generated and verified using the same secret key. This implies that the sender and receiver of a message must agree on the same key before initiating communications, as is the case with symmetric encryption. For the same reason, MAC does not provide the property of non-repudiation offered by signatures. Any user who can verify a MAC is also capable of generating MAC for other messages. In this situation user who sent the message later on can refuse that he had sent the message. MAC also faces key exchange problem. There are various cryptographic hash function based MAC algorithms such as Hash Message Authentication Code (HMAC) [5, 6], the envelope MAC scheme [7], and MDx-MAC [8]. HMAC is the most commonly used hash-based MAC. HMAC is a hash transformation parameterized with a secret key. Given the hash function \( H \), HMAC works roughly as follows. Let the key be \( K \), and let \( ipad \) and \( opad \) be distinct constant strings of the same length as \( K \). Then HMAC is of the form:

\[
HMAC(text, K) = H(K \oplus opad || H(K \oplus ipad || text)),
\]

where \( text \) is the text to be hashed together with key \( K \).

(iii) Secure Socket Layer

Secure Socket Layer (SSL) [9] is the facto standard for communication on the Internet. SSL enables authentication and secure communication on the Internet. These protocols prevent eavesdropping, falsification, and message forgery. The handshake protocol in SSL uses a hash function to create a message authentication code. SSL provides server authentication to clients. During the SSL handshake, once the parts have negotiated the protocol, these have to select also the hash methods to use for authentication. The client can choose the type of hash function to use.
(iv) **Pseudorandom Function**
Generating random bit sequences is an important problem in cryptography. The security of many cryptographic systems depends on the generation of unpredictable bit sequences. Such sequences are used, for example, in stream ciphers, digital signature schemes, key materials of encryption schemes, in challenge-response identification systems, and in many other cryptographic protocols. Hash functions are often used as pseudo random functions. That is, they provide a deterministic mechanism for generating random seeming bit streams from some input source without disclosing any information about the input. One important application of these generators are key derivation techniques. Key derivation functions are used to construct a number of cryptographically strong keys from an initial bit string, which is expected to contain some randomness, but not necessarily to be suitable as a cryptographic key by itself. In other words, the key derivation function is expected to evenly distribute the entropy of the input string over the output, which will often be of a different length than the input. Secret keys are very important to guarantee the security of cryptographic protocols. Using one single secret key in many protocols would be a dangerous practice: using this key in inappropriate ways or in an insecure protocol only once would be enough to reveal it, therefore threatening the security of all the other protocols that use the same key. For this reason, the private keys used in protocols are usually derived using a hash function. Because the hash computation cannot be reversed and its output is perfectly random, the leakage of one particular secret does not disclose any partial information on the master key. A typical use is generating cipher keying material after a Diffie-Hellman exchange. IKE uses HMAC for this purpose, as does TLS [9]. In the Digital Signature Standard (DSS) [10], pseudorandom numbers are used extensively, e.g., to compute large primes and to compute a so-called per message secret number. In [11], several examples are described.

(v) **Data Integrity**
As noted above, hash functions can be used to produce fingerprints of files or messages. Sometimes, instead of digitally signing these fingerprints, the values are stored separately from the data. This permits later detection of changes to the original data caused by a malicious or an error prone communication channel. The scheme works by the user receiving data through some insecure or faulty communication channel. The digest of the data is then computed and is compared to the digest of the original data sent by the trusted third party. One system in which
this is used is Tripwire. Tripwire is used as host intrusion detection system. Critical system files are fingerprinted; at intervals thereafter, the stored fingerprints are compared to values newly-calculated on the running system. If the message is tampered with, the digital fingerprint will change to reflect changes in the content. Therefore, the properties of cryptographic hash functions can be used to verify that files have not been altered; one can quickly determine data integrity. Notice though that one cannot determine specifically what contents of the message have changed, only that something in the file has been changed.

(vi) Time Stamping

Timestamped documents are at the heart of the legal use of electronic documents. To know when a document has been signed for example, the timestamp also is included in the message to be hashed. The Time Stamping protocol provides a time stamp token which guarantees the time at which data existed. A digital timestamp is sort of digital stamp used to prove the existence of a digital document at a certain date. The creation date of digital documents can be modified and go undetected. Thus, the creation date on a digital document is simply not reliable as a proof of the document's existence on the date that document claims to have been created. However, forward-dating is less attractive than back-dating for the simple fact that a conscious person will not believe that a document presented to him/her today was in fact created tomorrow. To avoid date of creation conflict, it is required that a Trusted Third Party (TTP) playing the role of a Time Stamping Authority (TSA) processes all valid digital time-stamping. Multiple time-stamping authorities can be contacted to increase the level of credibility of a document one wishes to timestamp. The process of creating a timestamp relies on digital signature scheme and hash functions, so does its security. Hash function to ensure the integrity of the time stamp.

(vii) Digital Image Watermarking

Digital watermark is a perceptually invisible pattern embedded in a digital image. The watermark can carry information about the owner of the image or the recipient, the image itself (watermarking for tamper detection and authentication), or some additional information accompanying the image (image caption embedding). Hash functions are also used in digital image watermarking. Another important application of hash functions is found in digital image watermarking. This serves the purpose of facilitating the detection of image manipulation. The original structure of the watermark utilized by a sender is expected to be known by the recipient.
If the watermarked image is manipulated, the watermark is also affected; hence when a recipient extracts the watermark, it will be different from that which is expected. It is a clever means of providing information security in situations where the information is an image. Hash functions are utilized in the watermarking algorithms; a common example is the Wong’s watermarking algorithm [12]. In Wong’s algorithm, the image is divided into blocks of pixels and each block is hashed independently. This is done when embedding the watermark and also when extracting it for verification purposes. When embedding the watermark, the hash output is truncated to a desired size and then it is XORed with a binary watermark to give the output that will be encrypted and placed in the corresponding least significant bit positions of the pixels of the image. Since the image can contain many blocks of pixels, it is imperative for the hash function to be able to operate at high speed and in a computationally efficient manner.

(viii) **Password Protection**
A standard example that well illustrates the use of cryptographic hash functions is password storage in a multiple-user system, where each user has to authenticate themselves by entering a password. The system has to check whether an entered password is correct, but simply storing all username-password matches in a file is not a good idea because such a file could easily be compromised by an attacker, who would then have access to all the passwords. What is done in practice is that the hash values of passwords are stored. That way an attacker who gains access to the file still does not know a single password. He/she cannot calculate them because the cryptographic hash function is oneway. But when a password is entered, its hash can be re-calculated and compared to the stored value, thus authenticating a user who knows the correct password. A related application is password verification. Passwords are usually not stored in clear text, for obvious reasons, but instead in digest form. To authenticate a user, the password presented by the user is hashed and compared with the stored hash. For both security and performance reasons, most digital signature algorithms specify that only the digest of the message be signed, not the entire message.

(ix) **Commitment Scheme**
In a commitment scheme that uses a cryptographic hash function, the user committing to a value will compute the digest of the value and send it to a trusted third party. This third party is often a newspaper or other public media. The user committing to the value can verify that no change
occurred during the publishing stage. The user verifying the commitment is able to record the digest of the value that has been committed. At a later stage the value is revealed and the verifier can recompute the digest and compare it to the recorded digest. A typical use of a cryptographic hash would be as follows: Alice poses to Bob a tough math problem and claims she has solved it. Bob would like to try it himself, but would yet like to be sure that Alice is not bluffing. Therefore, Alice writes down her solution, appends a random nonce, computes its hash and tells Bob the hash value (whilst keeping the solution secret). This way, when Bob comes up with the solution himself a few days later, Alice can verify his solution but still be able to prove that she had the solution earlier. In actual practice, Alice and Bob will often be computer programs, and the secret would be something less easily spoofed than a claimed puzzle solution. The above application is called a commitment scheme. Commitment schemes have been first introduced by Brassard et al. [13]; they have used for electronic coin flipping, zero-knowledge proofs of knowledge and in verifiable secret sharing. Commitment schemes can be constructed from pseudorandom number generators and from oneway functions.
APPENDIX  B

Source Code for Proposed Hash Functions

B.1 Source Code for MDA-192

package shaPkg;

public class MDA192 {

    // initial blocks used in each step function
    private int block[] = new int[16];
    private int blockIndex;
    private boolean NSA = true;
    // Constants A,B,C,D,E,F chaining variables;
    private int state[] = new int[6];
    private int dd[] = new int[6];
    private long count;
    private byte digestBits[];

    public MDA192(boolean b) {
        count = 0;
        NSA = b;
        /* MDA192 initialization constants */
        state[0] = 0x67452301;
        state[1] = 0xEFCDAB89;
        state[2] = 0x98BADCFE;
        state[3] = 0x10325476;
        state[4] = 0xC3D2E1F0;
        state[5] = 0x50A28BE6;
        digestBits = new byte[24];
        blockIndex = 0;
    }
}
public void printCount()
{
    System.out.println("Count=\"+count);"
}

public void printBlockIndex()
{
    System.out.println("BlockIndex=\"+blockIndex);"
}

public void finish() {
    byte bits[] = new byte[8];
    int i, j;

    for (i = 0; i < 8; i++) {
        bits[i] = (byte)((count >>> (((7 - i) * 8))) & 0xff);
    }
    update((byte) 128);

    while (blockIndex != 56)
    {
        update((byte) 0);
        for (i = 0; i < 8; i++) update(bits[i]);
    }

    for (i = 0; i < 24; i++) {
        digestBits[i] = (byte)((state[i>>2] >>> ((3-(i & 3)) * 8)) & 0xff);
    }
}

public void update(byte b) {
    int mask = (8 * (blockIndex & 3));
    count += 8;
}
//         printBlockIndex();
if (blockIndex == 64) {
    transform2();
    blockIndex = 0;
}
}

public void printStatus()
{
    System.out.println("Message Blocks");
    for (int i= 0; i < block.length; i++) {
        System.out.printf("Block[%d]=%d[Decimal],%x[Hex]\n",i,block[i],block[i]);
    }
}

private void transform2()
{
    /* Copy context->state[] to working vars */
    dd[0] = state[0];
    dd[1] = state[1];
    dd[2] = state[2];
    dd[3] = state[3];
    dd[4] = state[4];
    dd[5] = state[5];
    int a[] = {0,1,2,3,4,5};
    for (int i = 0; i < 96; i++) {
        if(i>=0&&i<=15)
            R0(dd,a[0],a[1],a[2],a[3],a[4],a[5],i);
else if (i>=16&&i<=23)
    R1(dd,a[0],a[1],a[2],a[3],a[4],a[5],i);
else if (i>=24&&i<=47)
    R2(dd,a[0],a[1],a[2],a[3],a[4],a[5],i);
else if (i>=48&&i<=71)
    R3(dd,a[0],a[1],a[2],a[3],a[4],a[5],i);
else if (i>=72&&i<=95)
    R4(dd,a[0],a[1],a[2],a[3],a[4],a[5],i);

for (int j = 0; j < a.length; j++) {
    a[j] = (a[j]+5)%6;
}

state[0] += dd[0];
state[1] += dd[1];
state[2] += dd[2];
state[3] += dd[3];
state[4] += dd[4];
state[5] += dd[5];

public int f1(int b, int c, int d)
{
    return (b & c) | ((~b) & d);
}

public int f2(int b, int c, int d)
{
    return b^c^d;
}
public int f3(int b, int c, int d)
{
    return (b & c) | (b & d) | (c & d);
}

/* Message Expansion */

public int blk16_51(int i)
{
    block[i & 15] =
        block[(i + 3) & 15] ^ block[(i + 8) & 15] ^ block[i & 15] ^
            (rol(block[(i + 1) & 15] ^ block[(i + 2) & 15] ^ block[(i + 15) & 15], 13));
    return block[i & 15];
}

public int blk52_95(int i)
{
    block[i & 15] =
            (rol(block[(i + 1) & 15] ^ block[(i + 2) & 15] ^ block[(i + 15) & 15] ^ block[(i + 20) & 15], 13));
    return block[i & 15];
}

public void R0(int data[], int a, int b, int c, int d, int e, int f, int i)
{
    int t;
    int temp = blk0(i);
    t = rol((data[f] + rol(data[a], 5) + f1(data[b], data[c], data[d]) + temp + 0x5A827999).temp % 32);
    data[a] = t;
    data[b] = data[a] ^ rol(temp, 15);
    data[c] = rol(data[b], 30);
    data[d] = data[b] ^ data[c];
    data[e] = data[c] ^ data[d];
    data[f] = data[d] ^ data[e];
}
public void R1(int data[], int a, int b, int c, int d, int e, int f, int i) {
    int t;
    int temp = blk16_35(i);
    t = rol((data[f] + rol(data[a], 5) + f1(data[b], data[c], data[d]) + temp + 0x5A827999) % 32);
    data[a] = t;
    data[b] = data[a] ^ rol(temp, 15);
    data[c] = rol(data[b], 30);
    data[d] = data[b] ^ data[c];
    data[e] = data[c] ^ data[d];
    data[f] = data[d] ^ data[e];
}

public void R2(int data[], int a, int b, int c, int d, int e, int f, int i) {
    int t;
    int temp;
    if (i >= 24 && i <= 35) {
        temp = blk16_35(i);
    } else {
        temp = blk36_95(i);
    }
    t = rol((data[f] + rol(data[a], 5) + f2(data[b], data[c], data[d]) + temp + 0x6ED9EBA1) % 32);
    data[a] = t;
    data[b] = data[a] ^ rol(temp, 15);
    data[c] = rol(data[b], 30);
    data[d] = data[b] ^ data[c];
    data[e] = data[c] ^ data[d];
    data[f] = data[d] ^ data[e];
}
public void R3(int data[], int a, int b, int c, int d, int e, int f, int i)
{
    int t;
    int temp;
    temp = blk36_95(i);
    t = rol((data[f] + rol(data[a], 5) + f3(data[b], data[c], data[d]) + temp + 0x8F1BBCDC) % 32);
    data[a] = t;
    data[b] = data[a] ^ rol(temp, 15);
    data[c] = rol(data[b], 30);
    data[d] = data[b] ^ data[c];
    data[e] = data[c] ^ data[d];
    data[f] = data[d] ^ data[e];
}
public void R4(int data[], int a, int b, int c, int d, int e, int f, int i)
{
    int t;
    int temp;
    temp = blk36_95(i);
    t = rol((data[f] + rol(data[a], 5) + f2(data[b], data[c], data[d]) + temp + 0xCA62C1D6) % 32);
    data[a] = t;
    data[b] = data[a] ^ rol(temp, 15);
    data[c] = rol(data[b], 30);
    data[d] = data[b] ^ data[c];
    data[e] = data[c] ^ data[d];
    data[f] = data[d] ^ data[e];
}
private final int rol(int value, int bits) {
    int q = (value << bits) | (value >> (32 - bits));
    return q;
}
private final int blk0(int i) {

block[i] = (rol(block[i],24)&0xFF00FF00) |
(rol(block[i],8)&0x00FF00FF);
    return block[i];
}
private final int blk(int i) {
    if(NSA)
    {
        block[i&15] = rol(block[i&15], 1);
    }
    return (block[i&15]);
}
public String digout() {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < 24; i++) {
        char c1, c2;
        c1 = (char) ((digestBits[i] >>> 4) & 0xf);
        c2 = (char) (digestBits[i] & 0xf);
        c1 = (char) ((c1 > 9) ? 'A' + (c1 - 10) : '0' + c1);
        c2 = (char) ((c2 > 9) ? 'A' + (c2 - 10) : '0' + c2);
        sb.append(c1);
        sb.append(c2);
    }
    return sb.toString();
}
B.2 Source Code for DSHA-1

/* Here source code for Random number generations and step updation are provided */
/* Random number generations */
public class randGen {
   public static int main(int k)
   {
      int i;
      int t,ytt=0,tt1;
      BigDecimal tt;
      double tempV,tempR,tempU;
      for(i=0;i<16;i++)
      {
         if(k<0)
            k=k*(-1);
         tempR=(double)Math.sqrt(2)*k;
         tempV = tempR;
         tempV = tempV - Math.floor((tempR)*(-1));
         tempU = tempV;
         tempU = tempU % 100000000;
         tt = BigDecimal.valueOf(tempU);
         ytt = tt.intValue();
         if(ytt < 10000000)
            ytt *= 10 ;
      }
      return ytt;
   }
   /* Step updation */
   public final void R0(int data[], int v, int w, int x, int y, int z, int i) {
      int block = f(i);
data[z] += ((data[w] & (data[x] ^ data[y])) ^ data[y]) + block + 0x5A827999 +
        ROL(data[v], 5);
    
data[z] ^= randGen.main(block);
    data[w] = ROL(data[w], 30);
}

public final void R1(int data[], int v, int w, int x, int y, int z, int i) {

    int block = g(i);
    data[z] += ((data[w] & (data[x] ^ data[y])) ^ data[y]) + g(i) + 0x5A827999 +
               ROL(data[v], 5);
    data[z] ^= randGen.main(block);
    data[w] = ROL(data[w], 30);
}

public final void R2(int data[], int v, int w, int x, int y, int z, int i) {

    int block = g(i);
    data[z] += (data[w] ^ data[x] ^ data[y]) + g(i) + 0x6ED9EBA1 +
               ROL(data[v], 5);
    data[z] ^= randGen.main(block);
    data[w] = ROL(data[w], 30);
}

public final void R3(int data[], int v, int w, int x, int y, int z, int i) {

    int block = g(i);
    data[z] += (((data[w] | data[x]) & data[y]) | (data[w] & data[x])) + g(i) + 0x8F1BBCDC +
               ROL(data[v], 5);
    data[z] ^= randGen.main(block);
    data[w] = ROL(data[w], 30);
}

public final void R4(int data[], int v, int w, int x, int y, int z, int i) {

    int block = g(i);
    data[z] += (data[w] ^ data[x] ^ data[y]) + g(i) + 0xCA62C1D6 +
               ROL(data[v], 5);
    data[z] ^= randGen.main(block);
    data[w] = ROL(data[w], 30);
}

152
/* Constant values */

int RANDOM_CONSTANTS[16];

unsigned int delta[16] =
{
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xdc8f6fa6, 0x9159eba1, 0xd2b88b29, 0x523b8e7e,
    0x9db62766, 0x6277a875, 0x80243107, 0xa9a2e18b
};

#define ROL(x, n) ( ( (x) << n ) | ( (x) >> (32-n) ) )
#define f(x) ( x ^ ROL(x,15) ^ ROL(x,27) )
#define g(x) ( x ^ (ROL(x,7) + ROL(x,25)) )

/*Step Functions for MNF-256*/
#define step_b1(A,B,C,D,E,F,G,H, M1,M2,D1,D2)
    temp1 = A + M1;
    temp2 = E + M2;
    A = temp1 + D1;
    E = temp2 + D2;
    temp1 = f(temp1);
    temp2 = g(temp2);
    temp3 = g(A);
    temp4 = f(E);
    B += temp1;
    F += temp2;
    C = (C + ROL(temp1, 13)) ^ temp3;
    G = (G + ROL(temp2, 3)) ^ temp4;
    D ^= ROL(temp3, 17);
    H ^= ROL(temp4, 8);
    A ^= RANDOM_CONSTANTS[0];
    B ^= RANDOM_CONSTANTS[2];
C ^= RANDOM_CONSTANTS[4]; \
D ^= RANDOM_CONSTANTS[6]; \
E ^= RANDOM_CONSTANTS[8]; \
F ^= RANDOM_CONSTANTS[10]; \
G ^= RANDOM_CONSTANTS[12]; \
H ^= RANDOM_CONSTANTS[14]; \
printf("B_1:%lx %lx %lx %lx %lx %lx\n",A,B,C,D,E,F,G,H);

//define step_b2(A,B,C,D,E,F,G,H, M1,M2,D1,D2)

temp1 = A + M1; \
temp2 = E + M2; \
A = temp1 + D1; \
E = temp2 + D2; \
temp1 = f(temp1); \
temp2 = g(temp2); \
temp3 = g(A); \
temp4 = f(E); \
B += temp1; \
F += temp2;
C = (C + ROL(temp1, 13)) ^ temp3; \
G = (G + ROL(temp2, 3)) ^ temp4; \
D ^= ROL(temp3, 17); \
H ^= ROL(temp4, 8); \
A ^= RANDOM_CONSTANTS[15]; \
B ^= RANDOM_CONSTANTS[13]; \
C ^= RANDOM_CONSTANTS[11]; \
D ^= RANDOM_CONSTANTS[9]; \
E ^= RANDOM_CONSTANTS[7]; \
F ^= RANDOM_CONSTANTS[5]; \
G ^= RANDOM_CONSTANTS[3]; \
H ^= RANDOM_CONSTANTS[1];

154
printf("B_2:%lx %lx %lx %lx %lx %lx %lx\n",A,B,C,D,E,F,G,H);

//-------------------------------
#define step_b3(A,B,C,D,E,F,G,H, M1,M2,D1,D2)
    temp1 = A + M1; \n    temp2 = E + M2; \n    A = temp1 + D1; \n    E = temp2 + D2; \n    temp1 = f(temp1); \n    temp2 = g(temp2); \n    temp3 = g(A); \n    temp4 = f(E); \n    B += temp1; \n    F += temp2; \n    C = (C + ROL(temp1, 13)) ^ temp3; \n    G = (G + ROL(temp2, 3)) ^ temp4; \n    D ^= ROL(temp3, 17); \n    H ^= ROL(temp4, 8); \n    A ^= RANDOM_CONSTANT[14]; \n    B ^= RANDOM_CONSTANT[12]; \n    C ^= RANDOM_CONSTANT[10]; \n    D ^= RANDOM_CONSTANT[8]; \n    E ^= RANDOM_CONSTANT[6]; \n    F ^= RANDOM_CONSTANT[4]; \n    G ^= RANDOM_CONSTANT[2]; \n    H ^= RANDOM_CONSTANT[0]; \nprintf("B_3:%lx %lx %lx %lx %lx %lx %lx\n",A,B,C,D,E,F,G,H);
static void FORK256_Mod(unsigned int *CV, unsigned int *M)
{
    CV[0] = 0x6a09e667;
    CV[1] = 0xbb67ae85;
unsigned long R1[8], R2[8], R3[8];
unsigned long temp1, temp2, temp3, temp4;
R1[0] = R2[0] = R3[0] = CV[0];

// BRANCH1(CV, M)
step_b1(R1[0], R1[1], R1[2], R1[3], R1[4], R1[5], R1[6], R1[7], M[0], M[1], delta[0], delta[1]);
step_b1(R1[7], R1[0], R1[1], R1[2], R1[3], R1[4], R1[5], R1[6], M[2], M[3], delta[2], delta[3]);
step_b1(R1[6], R1[7], R1[0], R1[1], R1[2], R1[3], R1[4], R1[5], M[4], M[5], delta[4], delta[5]);
step_b1(R1[5], R1[6], R1[7], R1[0], R1[1], R1[2], R1[3], R1[4], M[6], M[7], delta[6], delta[7]);
step_b1(R1[4], R1[5], R1[6], R1[7], R1[0], R1[1], R1[2], R1[3], M[8], M[9], delta[8], delta[9]);
step_b1(R1[3], R1[4], R1[5], R1[6], R1[7], R1[0], R1[1], R1[2], M[10], M[11], delta[10], delta[11]);
step_b1(R1[2], R1[3], R1[4], R1[5], R1[6], R1[7], R1[0], R1[1], M[12], M[13], delta[12], delta[13]);
step_b1(R1[1], R1[2], R1[3], R1[4], R1[5], R1[6], R1[7], R1[0], M[14], M[15], delta[14], delta[15]);

// BRANCH2(CV, M)
step_b2(R2[0], R2[1], R2[2], R2[3], R2[4], R2[5], R2[6], R2[7], M[14], M[15], delta[15], delta[14]);
step_b2(R2[7], R2[0], R2[1], R2[2], R2[3], R2[4], R2[5], R2[6], M[11], M[9], delta[13], delta[12]);
step_b2(R2[6], R2[7], R2[0], R2[1], R2[2], R2[3], R2[4], R2[5], M[8], M[10], delta[11], delta[10]);
step_b2(R2[5], R2[6], R2[7], R2[0], R2[1], R2[2], R2[3], R2[4], M[3], M[4], delta[9], delta[8]);
step_b2(R2[4], R2[5], R2[6], R2[7], R2[0], R2[1], R2[2], R2[3], M[2], M[13], delta[7], delta[6]);
step_b2(R2[3],R2[4],R2[5],R2[6],R2[7],R2[0],R2[1],R2[2],M[0],M[5],delta[5],delta[4]);
step_b2(R2[2],R2[3],R2[4],R2[5],R2[6],R2[7],R2[0],R2[1],M[6],M[7],delta[3],delta[2]);
step_b2(R2[1],R2[2],R2[3],R2[4],R2[5],R2[6],R2[7],R2[0],M[12],M[1],delta[1],delta[0]);

// BRANCH3(CV,M)
step_b3(R3[0],R3[1],R3[2],R3[3],R3[4],R3[5],R3[6],R3[7],M[7],M[6],delta[1],delta[0]);
step_b3(R3[7],R3[0],R3[1],R3[2],R3[3],R3[4],R3[5],R3[6],M[10],M[14],delta[3],delta[2]);
step_b3(R3[6],R3[7],R3[0],R3[1],R3[2],R3[3],R3[4],R3[5],M[13],M[2],delta[5],delta[4]);
step_b3(R3[5],R3[6],R3[7],R3[0],R3[1],R3[2],R3[3],R3[4],M[9],M[12],delta[7],delta[6]);
step_b3(R3[4],R3[5],R3[6],R3[7],R3[0],R3[1],R3[2],R3[3],M[11],M[4],delta[9],delta[8]);
step_b3(R3[3],R3[4],R3[5],R3[6],R3[7],R3[0],R3[1],R3[2],M[15],M[8],delta[11],delta[10]);
step_b3(R3[2],R3[3],R3[4],R3[5],R3[6],R3[7],R3[0],R3[1],M[5],M[0],delta[13],delta[12]);
step_b3(R3[1],R3[2],R3[3],R3[4],R3[5],R3[6],R3[7],R3[0],M[1],M[3],delta[15],delta[14]);

CV[0] = CV[0] + ((R1[0] + R2[0]) ^ (R3[0] + R2[0]));

printf("CV[FINAL HASH VALUE:MNQ-256-MODIFIED]\n%\x %\x %\x %\x %\x %\x %\x
%\x\n",CV[0],CV[1],CV[2],CV[3],CV[4],CV[5],CV[6],CV[7]);

} /* Random number generations */
#define a 16807 /* multiplier */
#define m 2147483647L /* 2**31 - 1 */
#define q 127773L /* m div a */
#define r 2836 /* m mod a */

long nextlongrand(long seed) {
}
unsigned long lo, hi;

lo = a * (long)(seed & 0xFFFF);
hi = a * (long)((unsigned long)seed >> 16);
lo += (hi & 0x7FFF) << 16;
if (lo > m)
{
    lo &= m;
    ++lo;
}
lo += hi >> 15;
if (lo > m)
{
    lo &= m;
    ++lo;
}
return (long)lo;

static long randomnum = 1;
long longrand(void) /* return next random long */
{
    randomnum = nextlongrand(randomnum);
    return randomnum;
}
LIST OF PUBLICATIONS


