CHAPTER- 3

DESIGN AND ANALYSIS OF A NEW CRYPTOGRAPHIC HASH FUNCTION-
“R-U HASH”
Traditionally, hash functions were designed in the keyless manner, where a hash function accepts a variable length input message and produces a fixed length digest as output. However, over the years, few significant weaknesses were found in some popular keyless hash functions. These weaknesses have motivated the researchers to begin considering the dedicated-key setting, which also allows for more rigorous security arguments. Moreover, it was also observed that existing keyless hash functions cannot be converted into a dedicated-keyed hash functions, because such keyless hash functions normally do not accommodate extra component-key in them. In this chapter, we the design of a new hash function algorithm with integration of a key. It serves the requirements of message integrity and source authentication both. The proposed algorithm offers features of simplicity as well as speed while implementing on processors of different bits.

### 3.1 Properties of a general hash algorithm

Properties of hash functions are generally decided as per their implementation requirement. For example, this is always advisable to design a hash function which can be implemented easily. For this, for any given message the hash computation must be easy enough. At the same time, the hash function should be able to compress the message information correctly. Other properties are driven from the cryptographic environment requirements [134]. We may define three major properties for secure and efficient hash function:

(a) Preimage Resistance—“Pre-image resistance property means that if provided a hash h it should be practically not possible to find any message x such that h = hash(x). A message \( \{0, 1\}^* \rightarrow \{0, 1\}^n \) is always found to be pre-image resistant if from given hash value \( v \in \{0,1\}^* \) it is not possible to find a message \( X \in \{0,1\}^* \) such that \( h(X) = v \) i.e. it should be practically not possible to obtain original message from supplied or predetermined hash value. Functions, that do not have this property, are susceptible to various attacks that come under the category of pre-image attacks. Function that contains this property is also known as one-way hash functions” [134].

---

(b) 2nd-Preimage Resistance—“Second Pre-image resistance means that if provided an input \(x\), it should be practically not possible to find any other input \(y\), with \(x \neq y\) having \(\text{hash}(x) = \text{hash}(y)\). A hash function \(h: \{0, 1\}^* \to \{0, 1\}^n\) is called second pre-image resistant if given a message \(M_1 \in \{0, 1\}^*\) such that \(\text{hash}(x) = \text{hash}(y)\). That is, there should be no two different messages for which we may find the same final hash value. The functions that do not have this property are vulnerable to second-pre-image attacks. This property is also known as weak collision resistance” [134].

(c) Collision Resistance—“Collision resistance means that it should be practically not possible to find any two different messages \(x\) and \(y\) that have same hash value. i.e. \(\text{hash}(x) = \text{hash}(y)\). A hash function \(\text{hash}: \{0,1\}^* \to \{0,1\}^n\) is called collision resistant if it is not possible to find two messages \(x\) and \(y\) \(\in \{0,1\}^*\) such that \(\text{hash}(x) = \text{hash}(y)\). i.e. the values of \(x\) and \(y\) colloid. Such a pair is called a cryptographic hash collision. It requires a hash value at least twice as long as that required for pre-image-resistance, otherwise collisions may be found by a birthday attack. This property is also known as strong collision resistance” [134].

The above stated properties make sure that any unauthorized intruder is not able to replace or change the input data without changing its fingerprint or hash value. Thus, whenever any two strings are found having the same message digest, we can very confidently claim that both strings are exactly the same [134].

### 3.2 Design of Proposed Hash Function

Typically any hash function has two components: a compression function and a construction. The compression function is a mapping function that transforms a larger arbitrary-size input to a smaller fixed-size output, and the construction is the method by which the compression function is being repeatedly called to process a variable-length message [135]. Traditionally hash functions are being designed without any usage of key component. However, many a few recent attacks have been successfully implemented on these traditional popular hash functions such as- SHA-1, MD5 etc. As we discussed in previous section, security of algorithm needs to be proved, most of the newly designed algorithms are based on previously established and accepted designs with few modifications. If established design promises few security aspects,
the new design will automatically do so. In the same line, this algorithm is also based on popular MD5 design principles. Furthermore, integration of key in each round of operation on individual blocks gives more strength to the proposed algorithm against many of the known attacks on MD5.

Currently, there are two methods being used to extend or strengthen the previously existing designs: First, to perform increased number of rounds of operations instead of prescribed number of functions in existing hashing algorithm (for example, instead of four primitive functions use more in case of MD5), or add some advanced coding or permutation steps (for example, use more scrambling techniques in SHA-1); second to increase the total buffer space and use different mixing step in each of the round. Building hash functions using block ciphers, as a base, is the most popular and most widely applied method. Hash functions that use this method, use a compression function that is like a block-cipher consisting of two inputs- a block of message and a key. At present, a protocol is considered strong and secure if it requires at least $2^{128}$ operations to perform attack on it. But however, it is definite that in near future there will be a need for stronger security protocols.

The proposed solution uses integration of keyed symmetric key block encryption algorithm in each step or round of hash compression function. Because symmetric key encryption algorithm works on use of single key, both sender and receiver use the same key. This common key may be shared between them using an encrypted link between them by Key Distribution Center (KDC).

It is the responsibility of KDC to send the common session key to both sender and receiver in communication. KDC uses master keys of both parties for this purpose. Because only these two parties carry their corresponding private keys, so no other user in the network may intercept and read the original message and make use of this session key. Except KDC and both parties, involved in the message transmission, no other user has any idea of the shared secret key. Thus, this method helps in validating identity of source as key with sender and receiver is now same.

In this solution, the working of hash compression function has been copulated with keyed encryption function. The output of compression function in each block is further used as input for keyed operation. Compression function gives output of 128 bit long. And keyed function takes an input block of 64 bit at a time. Thus, first, the
output of compression function is divided into two equal sized blocks, with a length of 64 bit, then it operates two times: initially, with left 64 bits and then with right 64 bits. Then the keyed function (encryption function) is applied on both 64 bit blocks one by one. The final output is of 64 bit for each left and right part, making total of 128 bit. This overall 128 bit output is then be used as 128 bit CVq for compression function processing of next block of input [138].

Assign IV to CV₀
Assign (E (K,B₁) || E (K,B₂)) to CVq

here,

| IV = MD buffer Initialization value as set by given compression function
| E = Block encryption scheme
| B₁ = Left 64 bits from output of hash value of 512 bit block
| B₂ = Right 64 bits from output of hash value of 512 bit block
| K= key used for each block

The proposed algorithm may be described as given pseudo code:

Step 1: Start
Step 2: Put padding bits at the end of input message
Step 3: Put length of original message at the output of Step 2.
Step 4: Divide the output of Step 3 into L blocks of equal size. (512 bit blocks).
Step 5: Initialize 128 bit MD buffer
Step 6: Repeat Steps 7 to 11 for all 512 bit L blocks
Step 7: Calculate 128 bit hash value for the i<sup>th</sup> block
Step 8: Break output of Step 7 into two equal size blocks (64 bits each).
Step 9: Encrypt both blocks (outputs of Step7) using keyed block encryption function.
Step 10: Combine both 64 bit outputs calculated in Step 9.
Step 11: Use the output of Step 10 as CV for next 512 bit block.
Step 12: Transmit the final output of hashing of last block (L<sup>th</sup> block) as final hash value to the receiver.
Step 13: End

---

Following figure 3.1 depicts the overall processing as per this scheme [138]

Most of the existing hash functions are fast and one-way hash functions, and they provide security in case adversary modifies data in unauthenticated manner. But it has been observed that only message integrity does not guarantee sufficient security against all of the proven attacks. For example, if the algorithm for generating the code is known, an adversary can generate the correct code after modifying the data, thus, ordinary error detecting codes are not adequate. Intentional modification is undetectable with such codes. That is, suppose a message X is sent by sender along with its calculated hash value h. After interruption, intruder fetches the message, and changes this X into X’. At the same time, he may also get copy of h and recalculates new hash h’ for new message X’, and then transmits it to the receiver. At the receiving end, the receiver recalculates hash on received version of message i.e. X’. Now, it will result in verified one, which is not true [136].

---

However, simple symmetric encryption techniques can be used to produce a cryptographic checksum. This checksum can be used to protect against unauthorized data modification which may be either accidental or intentional. Additionally, if a hashing scheme is combined with a block cipher encryption algorithm, than it can be made more secured and stronger against attacks. The hash function $h$ is defined in such a manner that from the message $X$, $h(X)$ can be calculated easily, but if one knows $h(X)$, than it should be infeasible to find even one message $X$ that will generate this value. Moreover, calculating any other message $M'$ that produces the same hash value, i.e., $h(X) = h(X')$, must also be infeasible. The hash value may then be given to any strong block encryption function, whose key is already known to sender and receiver both. The sender will use this key for calculating hash at its end. And the same corresponding key will be used by the receiver to revert the transformation and restore the value $h(X)$. At the receiving end, the function $h$ is applied to the received message $X$, and then he compares two values of $h(X)$. Only if the message is original and not modified after generating hash by sender, these two hash values will appear to be equal [136].

Designing and implementing a new secure hash function majorly includes fundamental two constructs- first, a compression function that may operate on any given input string in the form of block of data of a fixed length and then second to use the another function in cascade fashion so that output of compression function may extend the output length up to the string of arbitrary length. Our Design principal for the algorithm may be stated as: “make use of already proven techniques and build stronger one.” For this [137] -

- We base our hash function around two established techniques- compression function and keyed function.
- The compression function accomplishes the requirement of providing basic building block for hash algorithm.
- To combine source authentication along with message integrity, make use of any keyed function.
- The proposed technique gives a solution for unauthenticated changes made in the input message and receiver does not receive it anymore on the assumption that it is coming from original sender only.
3.3 Steps involved in Proposed Keyed-Hash Algorithm- “R-U Hash”

We describe the algorithmic design of proposed hash function as follows: [137] 4

Let us assume an input message M of length b bits. We will use following notations in the description of algorithms:

- + : addition modulo $2^{32}$
- $<<<$ S: circular left shift by S bit positions
- $\wedge$: bit-wise AND
- $\lor$: Bit-wise OR
- $\oplus$: bit-wise XOR
- $\neg$: bit-wise complement

The proposed algorithm may be divided into two phases- preprocessing and hash calculation.

The preprocessing phase is very much similar to that of MD-5 and SHA-1, involving padding and message length and further obtaining in m-blocks, each block of 512 bit length. The hash calculation is done on each 512 bit block in iterative manner in second phase of the algorithm. This phase also makes use of two 64 bit keys. The 512 bits are then compressed into 128 bits and provided as input for processing of next block of message. The output of processing of final block of message is called as message digest or hash value. The compression function makes use of S-Box, XOR, addition modulo $2^{32}$ and look-up tables. T

The use of primitive logical functions, which are implemented on hardware are readily available with a look-up table that helps in increasing the speed of hash function processing. Following are the few steps of proposed algorithm:

**Step 1: Pad the Original Message-**

The original message is padded by specific number of bits, so that the length of input message after padding becomes congruent to 448 modulo 512 (length $\equiv 448 \mod 512$).

---

512). For this purpose, first bit is always 1 and remaining bits are always 0. This is a compulsory step and thus, 1 to 512 bits may be appended, depending upon the length of original message.

**Step 2: Append Message Length**

After padding, length of original message is now put at the end of the result of step 1. This length is in 64 bit representation. After this step, the length of message is now in multiples of 512.

**Step 3: Initialize Buffer**

The algorithm uses a 128 bit buffer (four distinct words B1, B2, B3 and B4, 32 bit each), which is initialized with following hexadecimal values:

\[
\begin{align*}
B1 &= 01234567 \\
B2 &= 89ABCDEF \\
B3 &= FEDCBA98 \\
B4 &= 76543210
\end{align*}
\]

This step is done only for once, and then after receiving the output from first block acts as buffer for second block and so on. The final result of hashing is also stored in this.

**Step 4: Initialize t-table**

A 64 element t-table is used in the algorithm, which is prepared by following formula for each t value (ranging from 0 to 63):

\[K_t = \lfloor 2^{32} | \sin (t+1) \rfloor\]

where, t is in radians.

**Step 5: Four Secondary Functions**

The Algorithm makes use of four rounds. Each round uses different secondary logical function. These four secondary functions, known as: f1, f2, f3 and f4. Each function produces 32 bit word from 32 bit input word. The functions take 16 values from the previously discussed t-table-

\[
\begin{align*}
f_1 (B2, B3, B4) &= (B2 \land B3) \lor (\neg B2 \land B4) \text{ for } t = 0, \ldots, 15 \\
f_2 (B2, B3, B4) &= (B2 \land B4) \lor (B3 \lor \neg B4) \text{ for } t = 16, \ldots, 31 \\
f_3 (B2, B3, B4) &= (B2 \oplus B3 \oplus B4) \text{ for } t = 32, \ldots, 47
\end{align*}
\]
Step 6: Order of words for processing:

The processing is done in 4 rounds. Each round has 16 individual steps. For each step in each round, following sequence of words is used for processing.

- Round no. 1: \( j_0, j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, j_{10}, j_{11}, j_{12}, j_{13}, j_{14}, j_{15} \)
- Round no. 2: \( j_{16}, j_{17}, j_{18}, j_{19}, j_{20}, j_{21}, j_{22}, j_{23}, j_{24}, j_{25}, j_{26}, j_{27}, j_{28}, j_{29}, j_{30}, j_{31} \)
- Round no. 3: \( j_{32}, j_{33}, j_{34}, j_{35}, j_{36}, j_{37}, j_{38}, j_{39}, j_{40}, j_{41}, j_{42}, j_{43}, j_{44}, j_{45}, j_{46}, j_{47} \)
- Round no. 4: \( j_{48}, j_{49}, j_{50}, j_{51}, j_{52}, j_{53}, j_{54}, j_{55}, j_{56}, j_{57}, j_{58}, j_{59}, j_{60}, j_{61}, j_{62}, j_{63} \)

Processing is done in blocks. Each block is of 512 bit in length. A word is of 32 bits, thus, each block is made up of 16 words \((32 \times 16 = 512)\).

Step 7: Shifting-

Shifting is done in following amounts:

- Round no. 1: \( S_0 = 7, S_1 = 12, S_2 = 17, S_3 = 22, S_4 = 7, S_5 = 12, S_6 = 17, S_7 = 22, S_8 = 7, S_9 = 12, S_{10} = 12, S_{11} = 22, S_{12} = 7, S_{13} = 12, S_{14} = 17, S_{15} = 22 \)
- Round no. 2: \( S_{16} = 5, S_{17} = 9, S_{18} = 14, S_{19} = 20, S_{20} = 5, S_{21} = 9, S_{22} = 14, S_{23} = 20, S_{24} = 5, S_{25} = 9, S_{26} = 14, S_{27} = 20, S_{28} = 5, S_{29} = 9, S_{30} = 14, S_{31} = 20 \)
- Round no. 3: \( S_{32} = 4, S_{33} = 11, S_{34} = 16, S_{35} = 23, S_{36} = 4, S_{37} = 11, S_{38} = 16, S_{39} = 23, S_{40} = 4, S_{41} = 11, S_{42} = 16, S_{43} = 23, S_{44} = 4, S_{45} = 11, S_{46} = 16, S_{47} = 23 \)
- Round no. 4: \( S_{48} = 6, S_{49} = 10, S_{50} = 15, S_{51} = 21, S_{52} = 6, S_{53} = 10, S_{54} = 15, S_{55} = 21, S_{56} = 6, S_{57} = 10, S_{58} = 15, S_{59} = 21, S_{60} = 6, S_{61} = 10, S_{62} = 15, S_{63} = 21 \)

Step 8: Processing of message in sixteen 32-bit word (512 bit) blocks-

(a) for \( I = 0 \) to \( n-1 \) do  \( \text{ (here, } n= \text{ number of blocks) } \)
(b) divide $M_i$ into words $W_0, \ldots, W_{15}$ where $W_0$ is left most word.

(c) Initialization of 4 words $B_1 B_2 B_3 B_4$. Here each word is of 32 bit, i.e.

$$\text{total length} = 32 \times 4 = 128 \text{ bit.}$$

Assign $B_1$ to $B_1'$
Assign $B_2$ to $B_2'$
Assign $B_3$ to $B_3'$
Assign $B_4$ to $B_4'$

(’ represents new value of buffer word)

(d) For $t = 0$ to 63 do

Assign $B_2 + ((B_1 + f_t (B_2, B_3, B_4) + W_{jt} + K_t) \ll S_t$ to $X$
Assign $B_4$ to $B_1$
Assign $B_3$ to $B_4$
Assign $B_2$ to $B_3$
Assign $X$ to $B_2$

/* end of loop in step d*/

(e) Increment of 4 words $B_1 B_2 B_3 B_4$

Assign $B_1 + B_1'$ to $B_1$
Assign $B_2 + B_2'$ to $B_2$
Assign $B_3 + B_3'$ to $B_3$
Assign $B_4 + B_4'$ to $B_4$

(f) Make two 64 bit blocks $Y$ and $Z$ from $B_1 B_2 B_3 B_4$

Assign $B_1 B_2$ to $Y$
Assign $B_3 B_4$ to $Z$

(g) Generate 64 bit key for internal keyed operation. Out of these 64 bits, 8 are used as parity bits and rest 56 bits are used as effective key. Out of this single 56 bit key, 18 keys are generated, each of 48 bit long.

(h) Operations on $Y$ and $Z$ blocks- Both $Y$ and $Z$ are treated similarly. Each block is further subdivided into two partitions- left half of $Y$ block ($L_Y$) and right
half of Y block (R_y), and left half of Z block (L_z) and right half of Z block (R_z). Initially the right and left halves (R and L) are permuted (swapped), i.e.

Assign L to X
Assign R to L
Assign X to R

Now next L’ and R’ are produced as follows-

Assign R to L’
Assign L (+) f (R_{n-1}, K_n) to R’

Here, (=) is addition modulo 2^{32}.

This process is repeated for 16 times, each time with a different 48 bit key K.

Thus,
Assign R_{n-1} to L_n
Assign f(R_{n-1}, K_n) (+) L_{n-1} to R_n

After sixteenth round of operation, again perform final permutation (swapping of left and right half), thus,

Assign L_n to X
Assign R_n to L_n
Assign X to R_n

(here X is a 32 bit block used for permutation only.)

Final X= X XOR K_{17}
Final Y = Y XOR K_{18}

The algorithm for function f(R.K) is defined as follows-

a. Apply expansion permutation and output 48-bit data i.e assign E(R) to X.
b. XOR 48 bit output with the round key i. e. assign X ^ k to X’.
c. Apply S boxes function on X’ and generate output 32-bit data i. e. assign s(X’) to X”.
d. Apply the specific round permutation i. e. assign P(X”) to R’.
(i) Combine final 32 bit values of X and Y. After combining two 64 bit blocks, Y and Z respectively, we get one 128 bit block. These 128 bits are again stored in four 32 bit words B1B2B3B4.

/* end of loop in step i */

(j) After processing last 512 bit block, the final hash value is in B1B2B3B4, i.e. output is always 128 bit long digest.

We name this proposed hash function as “R-U Hash”.

The overall processing of proposed function can be shown as figure 3.2.

Figure 3.2: An illustrated view of processing of proposed hash function- “R-U Hash”, having keyed function in between (HF= hash compression function and EF= keyed function) [138]

The Encryption Function EF works on a block of 64 bit message, that’s why the 128-bit output of Hash Function $H_{\text{HF}}$ is subdivided into two parts, each of 64-bit. For the EF (Encryption Function), a key is used, with 64 bit length. Out of these 64 bits (eight 8-bit words) 8 bits are used as parity bits and remaining 56 bits are treated as actual effective key bits. The EF runs in 16 individual rounds with two permutation steps—one before first round and another after sixteenth round. For each round, a 48 bit key
is used out of 56-bit effective key. Thus, 16 sub-keys are created from one 56-bit key. As, JavaScript supports only 32-bit integers, we need to represent all 48 or 64-bit integers by two 32-bit integers in an array. There are 72 quadrillion or more possible combinations of the keys. And for each message (or for each message block) a unique key may be selected.

To create 16 individual sub-keys, we begin with conversion of the 8 character string, which represents the key, into two integers. Bits of this 8 character string are reorganized according to permuted choice1 (Perm1).

The permutation choice gives the following bits as per their position-

<table>
<thead>
<tr>
<th></th>
<th>57</th>
<th>49</th>
<th>41</th>
<th>33</th>
<th>25</th>
<th>17</th>
<th>9</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>50</td>
<td>42</td>
<td>34</td>
<td>26</td>
<td>18</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>51</td>
<td>43</td>
<td>35</td>
<td>27</td>
<td>19</td>
<td>11</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>52</td>
<td>44</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>55</td>
<td>47</td>
<td>39</td>
<td>31</td>
<td>23</td>
<td>15</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>54</td>
<td>46</td>
<td>38</td>
<td>30</td>
<td>22</td>
<td>14</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>53</td>
<td>45</td>
<td>37</td>
<td>29</td>
<td>21</td>
<td>13</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>20</td>
<td>12</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

This shows that after permutation, the first bit of the key is generated from 57th bit of the original key, second bit of key is generated from 49th bit of original key bit sequence, and so on. A specific permutation sequence is being used for permutation in this process. The permutation involves an intelligent way of rotation and switching of bits between the given two integers. For example, newint = (0x0f0f0f0f & (left >>> 4) ^ right) ; right = right ^ newint; left = left ^ (newint << 4); it results in rotation of 4x4 blocks of bits.

Now, we may call the initial 28 bits of Perm1 as X, and remaining 28 bits as Y. These two parts X and Y are then left shifted according to a specific pattern until all sixteen 48 bit sub-keys are produced. For example, X1 and Y1 are produced by left shifting X and Y by one bit place, X2 and Y2 are generated by left shifting X1 and Y1 by a further one place, and so on. These sixteen sub-keys (from X1 and Y1 to X16 and Y16) are further used to create permutation choice 2 (Perm2). Perm2 is a random bit
sequence. For Perm2, the bits are permuted four at a time using the array perm. The array perm contains fourteen sub-arrays. Each sub-array corresponds to 4 bits out of 56 bit sub-key. Thus, we have sixteen individual elements in each of the sub-array. The particular array is looked up for group of four bits. Every four bits of a sub-key are looked up in the corresponding array and then the results are all added together.

If, during the process, we store the messages and their corresponding result as strings, instead of array, then we may achieve better speed while implementing the program using JavaScript.

### 3.4 Coding

The program is written in Java. There are following files-

- Cryptography.java
- randomFileGenerator.java
- TestCrypto.java
- Testhash.java

#### Cryptography.java

```java
import java.io.IOException;
import java.security.InvalidKeyException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
```
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;

public class Cryptography
{

    private static final String CRYPTOGRAPHY_ALGO_DES = "DES";
    private static Cipher cipher = null;
    private static DESKeySpec keySpec = null;
    private static SecretKeyFactory keyFactory = null;

    private static String encrypt(String inputString, String commonKey)
    throws InvalidKeyException, IllegalBlockSizeException,
        BadPaddingException
    {
        String encryptedValue = "";
        SecretKey key = getSecretKey(commonKey);
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] inputBytes = inputString.getBytes();
        byte[] outputBytes = cipher.doFinal(inputBytes);

        // System.out.println("ip "+inputBytes[0].toString());
        // System.out.println("ip length"+inputBytes.length);
        // System.out.println("op "+outputBytes[0].toString());
        // System.out.println("op length"+outputBytes.length);

        encryptedValue = new
BASE64Encoder().encode(outputBytes);

        return encryptedValue;
    }
}
public static String decrypt(String encryptedString, String commonKey)
    throws InvalidKeyException, IllegalBlockSizeException,
    BadPaddingException, IOException
{
    String decryptedValue = "";
    encryptedString = encryptedString.replace(' ', '+');
    SecretKey key = getSecretKey(commonKey);
    cipher.init(Cipher.DECRYPT_MODE, key);
    byte[] recoveredBytes = cipher.doFinal(new BASE64Decoder()
        .decodeBuffer(encryptedString));
    decryptedValue = new String(recoveredBytes);
    return decryptedValue;
}

private static SecretKey getSecretKey(String secretPassword)
{
    SecretKey key = null;
    try
    {
        cipher = Cipher.getInstance(CRYPTOGRAPHY_ALGO_DES);
        keySpec = new DESKeySpec(secretPassword.getBytes("UTF8"));
    }
RandomFileGenerator.java

import java.io.*;
import java.util.*;

class RandomFileGenerator {
    public static void main(String args[])
    {
        try {
            FileWriter fstream = new FileWriter("test_data.txt");
        }
        catch (Exception e) {
            e.printStackTrace();
            System.out.println("Error in generating the secret Key");
        }
        return key;
    }
}

KeyFactory = SecretKeyFactory.getInstance(CRYPTOGRAPHY_ALGO_DES);
    key = keyFactory.generateSecret(keySpec);
BufferedWriter out = new BufferedWriter(fstream);

String alphabet = "0123456 7890 qwer tyui op asdf g h jkl zxcvb bnm,. WERTY FGHJK CVBNM< IOP{ } ASDFGHJ CFGYU";

int N = alphabet.length();

Random r = new Random();

StringBuilder sb = new StringBuilder();

for (int i = 0; i < Integer.parseInt(args[0]); i++)
{
    // This will generate 1MB file
    sb.append(alphabet.charAt(r.nextInt(N)));
}

out.write(sb.toString());

out.close();

}

} catch(Exception e) {
{
    e.printStackTrace();
}}

TestCrypto.java

public class TestCrypto
public static final String DES_ENCRYPTION_KEY = "12345670";
/**
 * @param args
 */
public static void main(String[] args)
{
    Try
    {
        // String input = "The quick brown fox jumps over the lazy dog.";
        String input =
"12345678901234561234567890123456123456789012345612345678901234561234567890123456123456789012345612345678901234561234567890123456123456789012345612345678901234561234567890123456"
;
        String encrypted = Cryptography.encrypt(input,
                         DES_ENCRYPTION_KEY);
        System.out.println(encrypted);
        String decrypted = Cryptography.decrypt(encrypted,
                         DES_ENCRYPTION_KEY);
        System.out.println(decrypted);
    }
    catch(Exception e)
    {
    }
}
import java.io.*;
import java.util.*;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Formatter;
import java.util.Scanner;

public class TestMD5
{
    public static void main(String[] args)
    {
        Try
        {
            Scanner user_input = new Scanner(System.in);
            System.out.println("Press 1 if you want to enter text, Press 2 for entering a file path");
            String option = user_input.next();
            String s, text = ""; // user text
            if (option.equals("1"))
            {
                System.out.println("Enter text to hash");
                text = user_input.next();
            }
        }
    }
}
else if (option.equals("2"))
{
    // file path
    System.out.println("Enter file path");
    BufferedReader in = new BufferedReader(new FileReader (user_input.next ( )));
    while ((s = in.readLine()) != null)
    {
        text = text + s;
    }
    in.close();
}
else
{
    System.out.println("Invalid input. Exit");
    System.exit(0);
}
long startTime = System.nanoTime();
System.out.println("Data length is "+text.length());
System.out.println("Encrypted key:\n"+getMD5HashVal(text));
long endTime = System.nanoTime();
System.out.println("Computation took "+(endTime - startTime)/1000000000.0 + " secs ");
}
catch(Exception e) {
    e.printStackTrace();
}

public static String getMD5HashVal(String strToBeEncrypted) {
    String encryptedString = null, str = "", encryptedString1 = null;
    String DES_ENCRYPTION_KEY = "12345677";
    StringBuilder sb = new StringBuilder();
    byte[] bytesToBeEncrypted;
    try {
        for (int i = 0; i < strToBeEncrypted.length(); i += 8) {
            int end_limit = i + 7;
            if (strToBeEncrypted.length() < end_limit) {
                end_limit = strToBeEncrypted.length();
            }
            bytesToBeEncrypted = strToBeEncrypted.substring(i, end_limit).getBytes("UTF-8");
            MessageDigest md = MessageDigest.getInstance("MD5");
        }
    }
}
byte[] theDigest = md.digest(bytesToBeEncrypted);
Formatter formatter = new Formatter();
for (byte b : theDigest)
{
    formatter.format("%02x", b);
}
encryptedString = formatter.toString().toLowerCase();
String encryptedMD5 = Cryptography.encrypt(encryptedString, DES_ENCRYPTION_KEY);
sb.append(encryptedMD5);
bytesToBeEncrypted = sb.toString().getBytes("UTF-8");
MessageDigest md1 = MessageDigest.getInstance("MD5");
byte[] theDigest1 = md1.digest(bytesToBeEncrypted);
Formatter formatter1 = new Formatter();
for (byte b : theDigest1)
{
    formatter1.format("%02x", b);
}
encryptedString1 = formatter1.toString().toLowerCase();
}

catch (Exception e)
randomFileGenerator.java

package keyed_hash_generation;

import java.io.*;

import java.util.*;

class randomFileGenerator {

    public String content;

    public randomFileGenerator(String s) {
        try {
            FileWriter fstream = new FileWriter("test_data.txt");
            BufferedWriter out = new BufferedWriter(fstream);
            String alphabet = "0123456 7890 qwertyuiop asdfghjklzxcvbnm,. WERTY FGHJK CVBNM< IOP} ASDFGHJCFGYU";
            int N = alphabet.length();
            Random r = new Random();
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < Integer.parseInt(s); i++) {
                sb.append(alphabet.charAt(r.nextInt(N)));
            }
            out.write(sb.toString());
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        randomFileGenerator generator = new randomFileGenerator("10");
        System.out.println(generator.content);
    }
}
3.5 Execution of R-U Hash Function

We have designed two user interfaces for implementing the design. First is using NetBeans and another is using simple java codes.

Using Netbeans, the execution is as follows-

```java
{     // This will generate 1MB file
    sb.append(alphabet.charAt(r.nextInt(N)));
}
out.write(sb.toString());
content=sb.toString();
out.close();
}
catch(Exception e)
{
    e.printStackTrace();
}
}
```

![Image](image-url)

*Figure 3.3: Execution of Proposed Design – “R-U Hash” (method-1)*
This opens a dialog box where we may enter the size of desired size of file in terms of bytes, and in turn it creates a random file of given size and calculates its keyed hash using a random key each time.

For example, a sample taken of a file of 2000 bytes-

![Image](image1.png)

**Figure 3.4: user interface of Proposed Design- “R-U Hash”**

Using simple java program, user is asked to either give a data to be hashed or path of file to be hashed, as given below-

![Image](image2.png)

**Figure 3.5 : Execution of Proposed Design- “R-U Hash” (method-2)**
Following is the sample of generating digest for same size file, here key is different each time.

![Image](image.png)

**Figure 3.6: A sample of Execution of Proposed Keyed Hash Function- “R-U Hash” (with time taken)**

The design of R-U hash function requires the size of the file to be given as input. It then generates the file of desired size. The file contains different data each time, no matter the name of file is same. Any specific file may also be given as input for this function. Each time a unique key is also being generated, which is being used in the function for generating the final digest.

A sample of random file, of specific size, is shown as below:
There is one more version of the proposed design, where user can either give a message or any existing file path for hashing. The code for that program is as follows:

RUHash.java

```java
import java.io.*;
import java.util.*;
import java.security.MessageDigest;
import java.util.Formatter;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```java
import java.io.*;
import java.util.*;
import java.security.MessageDigest;
import java.util.Formatter;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
```
public class RUHash
{
    public static void main(String[] args)
    {
        try
        {
            Scanner user_input = new Scanner(System.in);
            System.out.println("Press 1 if you want to enter text, Press 2 for entering a file path");
            String option = user_input.nextLine();
            String s, text = ""; // user text
            if (option.equals("1"))
            {
                System.out.println("Enter text to hash");
                text = user_input.nextLine();
            }
            else if (option.equals("2"))
            {
                // file path
                System.out.println("Enter file path");
                BufferedReader in = new BufferedReader(new FileReader(user_input.nextLine()));
                while ((s = in.readLine()) != null)
                {
                    text = text + s;
                }
            }
        }
    }
}
in.close();

}

else
{
   
   System.out.println("Invalid input. Exit");

   System.exit(0);

}

long startTime = System.nanoTime();

System.out.println("Data length is "+text.length());

System.out.println("Encrypted key:
"+getMD5HashVal(text));

long endTime = System.nanoTime();

System.out.println("Computation took "+(endTime - startTime)/1000000000.0 + " seconds");

}

catch(Exception e)
{
   
   e.printStackTrace();

}

}
MessageDigest md = null;

try {
    md = MessageDigest.getInstance("MD5");
} 

catch (NoSuchAlgorithmException ex) 
{
    Logger.getLogger(Md5des.class.getName()).log(Level.SEVERE, null, ex);
}

StringBuilder sb = new StringBuilder();

byte[] bytesToBeEncrypted =
    Character.toString(strToBeEncrypted.charAt(i)).getBytes("UTF-8");

//This contains 128 bits
byte[] bits_128 = md.digest(bytesToBeEncrypted);

Formatter str64BitsFormatter1 = new Formatter();

// 64 bits in first string
for (int k=0; k < strToBeEncrypted.length(); k++)
{
    str64BitsFormatter1.format("%02x", bits_128[k]);
String desString1 =
str64BitsFormatter1.toString().toLowerCase();

//Applying encryption on first half
String encryptedMD51 =
Cryptography.encrypt(desString1,
DES_ENCRYPTION_KEY);

Formatter str64BitsFormatter2 = new
Formatter();

// 64 bits in second string
for (int j=8; j < 15; j++)
{
    str64BitsFormatter2.format("%02x",
        bits_128[jj]);
}

String desString2 =
str64BitsFormatter2.toString().toLowerCase();

//Applying encryption on second half
String encryptedMD52 =
Cryptography.encrypt(desString2,
DES_ENCRYPTION_KEY);

//Combining the two DES Strings
String combined = encryptedMD51 +
encryptedMD52;

//Feeding that input to hash loop again
Formatter finalFormatter = new Formatter();
    for (byte b : finalDigest)
    {
        finalFormatter.format("%02x", b);
    }
    encryptedString = finalFormatter.toString().toLowerCase();
}
catch (Exception e)
{
    e.printStackTrace();
}
return encryptedString;
}
3.6 Security of Proposed Design for “R-U Hash”

All individual components of the proposed algorithm possess their respective security criteria, which ensure us that the algorithm is secure and collision free. We may give few arguments for security of the algorithm [136]:

- Mathematically secondary functions f1, f2, f3 and f4 are non-invertible and non-linear.
- If individual bits of B2, B3 and B4 are independent of each other, this implies that the overall bits of f(B2, B3, B4) are also independent of each other. It ensures the desired one-way property of hash function.
- During processing, in each of the round, accessing sequence for input words is different.
- In padding step, the padding is always done by 1 and then the number of zeros in the output of preprocessing phase to restrict the fixed point attacks in which the attacker’s main focus is to generate second pre-image or collisions by insertion of extra blocks into the input.
- In each step input is taken from the output of previous step. Thus, if we make changes at any one place in any one of the blocks, it will definitely affect the final output of algorithm. Thus, two different messages will never result in same output. It proves one more desirable hash property- the second pre-image resistance.
- Algorithm works on basic functions, such as modular arithmetic, XOR, addition, left shift, right shift, simple permutation etc. Thus, it does not require increased time requirement for processing.
- The required t-table and all 48 bit keys can be generated well in advance, so function need not wait in between for table element generation or key generation. This also helps in speeding up the execution of the algorithm.
- Use of XOR makes sure that output depends on all bits, rather than on neighboring ones only.
3.7 Key Generation and Usage

All popular existing keyed hash functions use either of two settings- Dedicated-Key setting or Integrated-Key setting. Both use fixed keys, i.e. once a key is dedicated, it will be used for each iteration of compression function. But, in the proposed solution, we will use 16 different key combinations in an iteration of compression function, individually on two word combinations Y and Z respectively.

Obviously this approach is more time consuming than keyless one, and it also increases overhead for computing hash by at least n * 2t, where n is total number of blocks and t is computation cost for one block either Y or Z. If we run it in parallel for both of these blocks simultaneously than computation time will increase by only n * t. Now, the efficiency lies in implementation of key function in hashing, and as we have already discussed earlier, because of simpler functions it has come out as a light weight function and will not take much time or efforts for whole message length.

3.8 Implementation

The proposed keyed hash function can be implemented efficiently on different platforms. The tables, codes, variables etc, used in the program, does not require large space. If we may use larger cache memory, then higher performance and better throughput can be achieved. This new hash function design uses the same building blocks as well tested and evaluated MD5 and DES, so we may expect similar performance and space characteristics. But as compared to MD5, SHA-0, SHA-1 etc., it provides more security because of implementation and use of key. Thus, it is a better message authentication code, which may take few milliseconds, more than MD5 or SHA-1 but at the same time is stronger and less vulnerable.

3.9 Performance Analysis of R-U Hash

Any hash function should confirm to the requirements as we stated earlier in this chapter and in previous chapters also. At the same time, the time taken by the
algorithm to perform hashing should not be more, as we know that information is no longer required if it is not given in required time frame. Thus, we tested the function for multiple size inputs, and found that its execution time not on higher side, further we compared it with other existing popular hash functions and found that the proposed hash function is taking either equal or lesser time as compared to them.

We have run and tested the function for a large number of inputs. Each time a different key is generated and results in different hash value even for the same input.

We executed this function for different input size (similar number of test cases for same input size between 1000 bytes to 50000 bytes), that is shown in Figure-3.9. It is found that average execution time is proportional to input size (Figure-3.10).
Following is the sample for which we tested it:

<table>
<thead>
<tr>
<th>Sample No.</th>
<th>Sample size</th>
<th>1000 Bytes</th>
<th>5000 Bytes</th>
<th>10000 Bytes</th>
<th>15000 Bytes</th>
<th>20000 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 Bytes</td>
<td>0.332673244</td>
<td>0.462357222</td>
<td>0.580623107</td>
<td>0.700160441</td>
<td>0.75854595</td>
</tr>
<tr>
<td>2</td>
<td>0 Bytes</td>
<td>0.340871827</td>
<td>0.46993603</td>
<td>0.591579854</td>
<td>0.734449234</td>
<td>0.759229619</td>
</tr>
<tr>
<td>3</td>
<td>5000 Bytes</td>
<td>0.339580214</td>
<td>0.468078054</td>
<td>0.585266399</td>
<td>0.686116634</td>
<td>0.760194583</td>
</tr>
<tr>
<td>4</td>
<td>10000 Bytes</td>
<td>0.343019806</td>
<td>0.468801054</td>
<td>0.588281373</td>
<td>0.682300649</td>
<td>0.751560039</td>
</tr>
<tr>
<td>5</td>
<td>15000 Bytes</td>
<td>0.344285335</td>
<td>0.473114527</td>
<td>0.590523316</td>
<td>0.686416065</td>
<td>0.841033585</td>
</tr>
<tr>
<td>6</td>
<td>20000 Bytes</td>
<td>0.342423785</td>
<td>0.468857452</td>
<td>0.583781072</td>
<td>0.680739658</td>
<td>0.764386492</td>
</tr>
<tr>
<td>7</td>
<td>1000 Bytes</td>
<td>0.342423785</td>
<td>0.466135899</td>
<td>0.58114636</td>
<td>0.680777803</td>
<td>0.763523771</td>
</tr>
<tr>
<td>8</td>
<td>15000 Bytes</td>
<td>0.34277462</td>
<td>0.465428712</td>
<td>0.592061805</td>
<td>0.679546775</td>
<td>0.75441323</td>
</tr>
<tr>
<td>9</td>
<td>20000 Bytes</td>
<td>0.341188306</td>
<td>0.465248853</td>
<td>0.584590008</td>
<td>0.683288169</td>
<td>0.743069423</td>
</tr>
<tr>
<td>10</td>
<td>1000 Bytes</td>
<td>0.34592443</td>
<td>0.47225321</td>
<td>0.58887875</td>
<td>0.688882888</td>
<td>0.760650966</td>
</tr>
<tr>
<td>11</td>
<td>5000 Bytes</td>
<td>0.340549989</td>
<td>0.464520458</td>
<td>0.589643473</td>
<td>0.69035002</td>
<td>0.760204528</td>
</tr>
<tr>
<td>12</td>
<td>10000 Bytes</td>
<td>0.342157066</td>
<td>0.469778743</td>
<td>0.587909787</td>
<td>0.687582061</td>
<td>0.756367481</td>
</tr>
<tr>
<td>13</td>
<td>15000 Bytes</td>
<td>0.342291356</td>
<td>0.466466648</td>
<td>0.588509281</td>
<td>0.688847962</td>
<td>0.807620163</td>
</tr>
<tr>
<td>14</td>
<td>20000 Bytes</td>
<td>0.334691404</td>
<td>0.465526786</td>
<td>0.599113771</td>
<td>0.800028101</td>
<td>0.75650966</td>
</tr>
<tr>
<td>15</td>
<td>1000 Bytes</td>
<td>0.34125231</td>
<td>0.468790756</td>
<td>0.584780232</td>
<td>0.683646052</td>
<td>0.760844649</td>
</tr>
<tr>
<td>16</td>
<td>5000 Bytes</td>
<td>0.341181324</td>
<td>0.455075503</td>
<td>0.593198293</td>
<td>0.691551685</td>
<td>0.753208993</td>
</tr>
<tr>
<td>17</td>
<td>10000 Bytes</td>
<td>0.343423201</td>
<td>0.468224153</td>
<td>0.58173765</td>
<td>0.68784374</td>
<td>0.757487397</td>
</tr>
<tr>
<td>18</td>
<td>15000 Bytes</td>
<td>0.340036787</td>
<td>0.467105252</td>
<td>0.597215259</td>
<td>0.685949628</td>
<td>0.771444106</td>
</tr>
<tr>
<td>19</td>
<td>20000 Bytes</td>
<td>0.355204174</td>
<td>0.464923026</td>
<td>0.584060167</td>
<td>0.684786554</td>
<td>0.745788911</td>
</tr>
<tr>
<td>20</td>
<td>1000 Bytes</td>
<td>0.342856739</td>
<td>0.46652975</td>
<td>0.587583143</td>
<td>0.690011085</td>
<td>0.754448318</td>
</tr>
<tr>
<td>21</td>
<td>5000 Bytes</td>
<td>0.335547418</td>
<td>0.463352366</td>
<td>0.587364102</td>
<td>0.686536796</td>
<td>0.750048266</td>
</tr>
</tbody>
</table>

Table 3.1: Time taken in seconds for hashing the file of given input size (1000-20,000 Bytes)
<table>
<thead>
<tr>
<th>Sample No.</th>
<th>25000 Bytes</th>
<th>30000 Bytes</th>
<th>35000 Bytes</th>
<th>40000 Bytes</th>
<th>45000 Bytes</th>
<th>50000 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.796541201</td>
<td>0.870928712</td>
<td>0.949689038</td>
<td>0.99974807</td>
<td>1.057885076</td>
<td>1.117469352</td>
</tr>
<tr>
<td>2</td>
<td>0.807235634</td>
<td>0.868958903</td>
<td>0.941026227</td>
<td>1.001175721</td>
<td>1.057171417</td>
<td>1.121867163</td>
</tr>
<tr>
<td>3</td>
<td>0.802225051</td>
<td>0.874140823</td>
<td>0.937519668</td>
<td>1.009684349</td>
<td>1.041272682</td>
<td>1.109476547</td>
</tr>
<tr>
<td>4</td>
<td>0.80932048</td>
<td>0.877931611</td>
<td>0.934373957</td>
<td>1.000985123</td>
<td>1.064755779</td>
<td>1.103416197</td>
</tr>
<tr>
<td>5</td>
<td>0.808562943</td>
<td>0.867072997</td>
<td>0.945257763</td>
<td>1.006128411</td>
<td>1.055749417</td>
<td>1.102975765</td>
</tr>
<tr>
<td>6</td>
<td>0.81847014</td>
<td>0.879596145</td>
<td>0.953985125</td>
<td>0.994965299</td>
<td>1.05112436</td>
<td>1.121372922</td>
</tr>
<tr>
<td>7</td>
<td>0.818606414</td>
<td>0.863960255</td>
<td>0.936540677</td>
<td>0.990444176</td>
<td>1.080804727</td>
<td>1.131000306</td>
</tr>
<tr>
<td>8</td>
<td>0.809647085</td>
<td>0.868425283</td>
<td>0.921923838</td>
<td>0.999616958</td>
<td>1.038913693</td>
<td>1.113085432</td>
</tr>
<tr>
<td>9</td>
<td>0.817465726</td>
<td>0.866870548</td>
<td>0.931137015</td>
<td>1.002703862</td>
<td>1.058542555</td>
<td>1.112133359</td>
</tr>
<tr>
<td>10</td>
<td>0.810994749</td>
<td>0.871677543</td>
<td>0.935372887</td>
<td>1.001870144</td>
<td>1.060740026</td>
<td>1.125126722</td>
</tr>
<tr>
<td>11</td>
<td>0.816945449</td>
<td>0.891525295</td>
<td>0.933138237</td>
<td>1.001931908</td>
<td>1.042748525</td>
<td>1.113411996</td>
</tr>
<tr>
<td>12</td>
<td>0.815483039</td>
<td>0.877013939</td>
<td>0.986801473</td>
<td>0.989039912</td>
<td>1.062434013</td>
<td>1.10022719</td>
</tr>
<tr>
<td>13</td>
<td>0.818223038</td>
<td>0.884060366</td>
<td>0.934745689</td>
<td>0.982780597</td>
<td>1.055889085</td>
<td>1.125198541</td>
</tr>
<tr>
<td>14</td>
<td>0.813667665</td>
<td>0.866222402</td>
<td>0.943821569</td>
<td>1.008631986</td>
<td>1.054917735</td>
<td>1.104885254</td>
</tr>
<tr>
<td>15</td>
<td>0.811468279</td>
<td>0.882963271</td>
<td>0.941723655</td>
<td>0.996459629</td>
<td>1.047721709</td>
<td>1.131061588</td>
</tr>
<tr>
<td>16</td>
<td>0.811198541</td>
<td>0.907709933</td>
<td>0.933254803</td>
<td>0.991503937</td>
<td>1.04203429</td>
<td>1.106922171</td>
</tr>
<tr>
<td>17</td>
<td>0.813745946</td>
<td>0.887033972</td>
<td>0.924838377</td>
<td>1.011198954</td>
<td>1.072295576</td>
<td>1.109893471</td>
</tr>
<tr>
<td>18</td>
<td>0.809550338</td>
<td>0.935912607</td>
<td>0.935002439</td>
<td>1.000259775</td>
<td>1.045856667</td>
<td>1.107799392</td>
</tr>
<tr>
<td>19</td>
<td>0.813248516</td>
<td>0.884661523</td>
<td>0.933699602</td>
<td>0.987944622</td>
<td>1.068559238</td>
<td>1.11485606</td>
</tr>
<tr>
<td>20</td>
<td>0.819525379</td>
<td>0.881440085</td>
<td>0.943515341</td>
<td>1.027753518</td>
<td>1.04273212</td>
<td>1.133299436</td>
</tr>
<tr>
<td>21</td>
<td>0.813432624</td>
<td>0.868711862</td>
<td>0.93149392</td>
<td>1.00757679</td>
<td>1.056605997</td>
<td>1.144456353</td>
</tr>
</tbody>
</table>

Table 3.2: Time taken in seconds for hashing the file of given input size (25,000-50,000 Bytes)
Figure 3.9: Time taken by R-U Hash for sample inputs

The suitable chart has been shown in Figure 3.10.

Figure 3.10: Average time taken by R-U Hash for sample inputs
The average time taken for these samples is shown in following table 3.3-

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Average time taken in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 Bytes</td>
<td>0.341636053</td>
</tr>
<tr>
<td>5000 Bytes</td>
<td>0.466690688</td>
</tr>
<tr>
<td>10000 Bytes</td>
<td>0.587992724</td>
</tr>
<tr>
<td>15000 Bytes</td>
<td>0.694276762</td>
</tr>
<tr>
<td>20000 Bytes</td>
<td>0.763360959</td>
</tr>
<tr>
<td>25000 Bytes</td>
<td>0.81216944</td>
</tr>
<tr>
<td>30000 Bytes</td>
<td>0.879848479</td>
</tr>
<tr>
<td>35000 Bytes</td>
<td>0.939469586</td>
</tr>
<tr>
<td>40000 Bytes</td>
<td>1.000590654</td>
</tr>
<tr>
<td>45000 Bytes</td>
<td>1.055178795</td>
</tr>
<tr>
<td>50000 Bytes</td>
<td>1.116663582</td>
</tr>
</tbody>
</table>

Table 3.3: Average time taken by R-U Hash for sample inputs

We tested the algorithm on number of inputs, where input is fixed to a definite size (say 1000 Bytes). In such a case, the execution time is almost same as shown in table 3.4 and figure 3.11, but each time key is not the same.

Different key is being generated even for the same input, thus, it results in different hash value each time.

As this function uses two basic rules: padding and fixed initialization vector. Thus, it is also safe from fixed point attack and second pre-image collision attack. At the same time, there exists no method for getting original message from hash value. It also uses the concept of key for generating hash value, which implies that there is no chance for adversary to compute hash value for a new message and to send it to receiver for the purpose of forging, because we assume that key is known to receiver and sender only, and it is delivered to them by trusted Key Distribution Center (KDC).
<table>
<thead>
<tr>
<th>Sample No.</th>
<th>Time Taken in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.333</td>
</tr>
<tr>
<td>2</td>
<td>0.341</td>
</tr>
<tr>
<td>3</td>
<td>0.340</td>
</tr>
<tr>
<td>4</td>
<td>0.343</td>
</tr>
<tr>
<td>5</td>
<td>0.344</td>
</tr>
<tr>
<td>6</td>
<td>0.342</td>
</tr>
<tr>
<td>7</td>
<td>0.342</td>
</tr>
<tr>
<td>8</td>
<td>0.343</td>
</tr>
<tr>
<td>9</td>
<td>0.341</td>
</tr>
<tr>
<td>10</td>
<td>0.346</td>
</tr>
<tr>
<td>11</td>
<td>0.341</td>
</tr>
<tr>
<td>12</td>
<td>0.342</td>
</tr>
<tr>
<td>13</td>
<td>0.342</td>
</tr>
<tr>
<td>14</td>
<td>0.335</td>
</tr>
<tr>
<td>15</td>
<td>0.341</td>
</tr>
<tr>
<td>16</td>
<td>0.341</td>
</tr>
<tr>
<td>17</td>
<td>0.343</td>
</tr>
<tr>
<td>18</td>
<td>0.340</td>
</tr>
<tr>
<td>19</td>
<td>0.345</td>
</tr>
<tr>
<td>20</td>
<td>0.343</td>
</tr>
</tbody>
</table>

Table 3.4: Time taken by different samples of equal size (1000 Bytes)
Security from various attacks:-

To perform brute force attack on an n-bit message digest an attacker must take about $2^n$ computations of hash function to obtain the pre-image with a significant percent of probability. Thus for R-U hash (a 128 bit digest) it takes minimum $2^{128}$ computations for an attacker before performing a successful brute force pre-image attack and brute force second pre-image attack. But as it also uses a 64 bit key, its required minimum calculations are $2^{192}$, which is better than MD-5, RIPEMD-160, SHA-1, SHA-0 etc. Similarly, it also takes $2^{96}$ computations for brute force collision attack.

But in practical it is not feasible to actually produce this huge amount of computational power today.

### 3.10 Improvements from Existing Hash Functions

Any hash function can be compared on the basis of any of the following two parameters- time taken for computation or memory occupied for computation that is time complexity or space complexity. Software implementation of the algorithm was tested on system with Intel based CPUs Pentium® -4 2.66 GHz with 1GB RAM. We
also run few of the existing hash functions on the same machine with same configuration.

We considered time aspect as major concern for comparison of this algorithm with existing algorithms, because lesser time requirement is one of the major criteria for selection of any specific hash function during message passing. As such, there is not any typical acceptable value of time considered for hash function; still it is assumed that function should not take longer time for calculation itself because hashing is like a required overhead while communication for security purpose.

Following table 3.5 presents time required for calculation of hash using various hash functions in contrast with R-U hash. We tested functions for 1 MB, 5 MB and 10 MB data files. It shows that the algorithm is the third fastest output after MD5 and RIPEMD-128. And we may argue in support of lesser speed as compared to MD5 and RIPEMD-128 with the fact that it is using a key in the algorithm and thus providing more security as compared to these algorithms [136].

<table>
<thead>
<tr>
<th>Hash Algorithm</th>
<th>time taken (in ms) for 1 MB data</th>
<th>time taken (in ms) for 5 MB data</th>
<th>time taken (in ms) for 10 MB data</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIPEMD-128</td>
<td>250</td>
<td>356</td>
<td>498</td>
</tr>
<tr>
<td>MD5</td>
<td>312</td>
<td>401</td>
<td>522</td>
</tr>
<tr>
<td>R-U Hash</td>
<td>346</td>
<td>480</td>
<td>592</td>
</tr>
<tr>
<td>SHA-512</td>
<td>422</td>
<td>530</td>
<td>632</td>
</tr>
<tr>
<td>SHA-256</td>
<td>437</td>
<td>541</td>
<td>654</td>
</tr>
<tr>
<td>SHA-224</td>
<td>453</td>
<td>561</td>
<td>687</td>
</tr>
</tbody>
</table>

Table 3.5: Comparison of software implementation of proposed algorithm with various known algorithms

This comparison shows that proposed hash design is worth using as it computes secure digest in a faster manner. It can also be taken in contrast with other existing hash functions as per their security property being provided.

The Figure 3.13 shows this comparison in pictorial format.
The table 3.5 and figure 3.13 show that as we increase the message size the time taken for computation increases in incremental order.

![Average time taken by various hash functions](image)

**Figure 3.13: Comparison of execution time taken by few existing hash functions and proposed hash function (X axis shows various hash functions and Y axis shows time taken in milliseconds)**

The table 3.6 differentiates the properties of existing hash functions with R-U hash. It is clearly shown the advancement and applicability of R-U hash from this table.

Thus, the proposed hash algorithm provides security, speed of computation, message integrity and source authentication, and it proved to be a practical and valuable function in the area of network security.
Properties of existing hash functions | Properties of R-U hash function
---|---
MD2, MD4, MD5, SHA family, RIPEMD, BLAKE, TAV etc message detection codes do not use “key”. Only message and initialization values are used as input to the function. | It uses “key” along with message and initialization values are used as input to the function.

MD2, MD4, MD5, SHA family, RIPEMD, BLAKE, TAV etc message detection codes provide only message integrity. | It provides message integrity and source authentication both.

Message Authentication codes such as HMAC and nested MAC use key but the simply prefix or postfix the key to the message, So it is easy to forge. | Key is used in each round of operation, not just pre or post fixed, which makes it difficult to forge.

In message authentication codes such as HMAC and nested MAC, key can easily be traced easily if message length is known in advance. | Key cannot be traced from final digest as it becomes integral part of hash function.

Collisions can be performed as only message and IV is put to the function. | To find collision is more difficult because of key.

Number of computations to perform Collisions and attacks depend on digest size only. Thus for 128 bit MD5 $2^{128}$ computations are required to perform brute force pre-image and brute force second pre-image attack. For RIPEMD-160, SHA-1, SHA-0, required calculations are minimum $2^{160}$. | Number of computations to perform Collisions and attacks depend on key size and digest size both. Thus for 128 bit R-U Hash $2^{192}$ computations are required to perform brute force attack.

**Table 3.6: Comparison of properties of existing hash functions with R-U Hash**