International Journal of Engineering Research & Management Technology

ISSN: 2348-4039

January-February-2023 Volume 10, Issue-1

Email: editor@ijermt.org www.ijermt.org

A STUDY ON CRYPTOGRAPHIC HASH FUNCTION- "R-U HASH"

Sudheer Kumar,

Research Scholar, Dept of Computer Application,

Maharaja Agrasen Himalayan Garhwal University

Dr Inderpal Singh,

Professor, Dept of Computer Application,

Maharaja Agrasen Himalayan Garhwal University

ABSTRACT

Cryptography is the science and art of information security. The science and art of information disclosure are cryptanalysis. Combining all of these concepts creates the term "cryptology," which refers to the science and art of concealing and revealing information. Cryptology was only concerned with information privacy up to the middle of the 1970s, such as that the information should not be copied, seen, read, or printed by any unauthorized parties. However, there are currently more goals for information security than just, well, secrecy. The cryptography controls the process of authenticating data coming from an outsider or enemy. Anyone who wants to damage the communication or message between two parties is the enemy. They are viewed as adversaries of secure communication. In order to transport messages from source to goal in a verifiable manner, cryptography basically has to resist the enemy's objective. To start, we must understand the goals of security. Modern cryptography relies heavily on cryptographic hash functions to maintain the integrity of messages. Any message can be passed into a hash function, which produces a hash value, hash result, hash code, digest, or just a hash. It can be defined simply as a function h that converts bit-strings of any finite length to strings of fixed length of n bits. Depending on the particular hash function being used, the output size of a hash typically ranges from 128 bits to 512 bits (digests fewer than 128 bits are regarded as insecure, while higher than 512 bits would generate more overhead during transmission). The term "n-bit hash" refers to the output of the hash function, which is an n-bit digest. The function is many-to-one for a domain (denoted as Do) and range (denoted as Ra) with function value HASH: Do Ra and |Do| > |Ra|. It suggests that we cannot avoid using the same hash value for input pairings that are distinct to one another. In fact, if we were to limit HASH to an m-bit input domain where t > n and HASH is supposed to be "random" by taking into account that all outputs are essentially equally likely, then there would

be about 2m-n inputs mapping to every output, and the probability of two randomly chosen inputs colliding to the same output is 2-n (independent of m). The underlying idea of hash functions in cryptography is that a digest is viewed as a miniature representation of an input string that may be used under the presumption that it is solely associated with that input string. The message digest is sometimes known as an imprint, a digest digital finger print, or just a little representative image.

KEY WORDS: Cryptography, HASH Functions, Computer Science, Random

INTRODUCTION

CRYPTOGRAPHIC HASH FUNCTION- "R-U HASH"

Hash functions have traditionally been created in a keyless way, where they accept messages of any length as input and output a fixed-length digest. However, over time, a few severe flaws in a few common keyless hash functions were discovered. These flaws have prompted the researchers to start thinking about the dedicated-key setting, which also enables stronger security justifications. Furthermore, it was discovered that keyless hash functions that already exist cannot be changed into dedicated-keyed hash functions because they typically do not include extra component-keys. We design a new hash function algorithm with key integration in this chapter. It fulfills both the requirements for source authentication and message integrity. While being implemented on processors of various bit sizes, the suggested method offers features of both speed and simplicity.

NONCONVENTIONAL HASHING APPROACHES

- **Construction of Hash Functions from Chaos Theory**

Recent chaos-based hash function proposals show an appealing design trend. These hash functions are based on chaos theory, which is a model for dynamic systems in mathematics. Initial value sensitivity, pseudorandomness, onewayness, and unpredictable orbit are characteristics of chaos systems that are also needed to create a hash function. Chaos theory-based hash algorithms use chaotic maps. Maps that display certain chaotic tendencies, such as logistical and tent maps, are called chaotic maps. Wong created a hashing algorithm based on the quantity of one-dimensional logistic map iterations. A one-way hash function built on the chaotic map and featuring a movable parameter was introduced by Xiao et al.. A hash function based on chaotic tent maps was proposed by Yi. A one-way hash function construction based on connected two-dimensional map lattices was presented by Wang et al. in. A one-way hash function construction based on a chaotic map network was proposed by Yang et

al.. Due to their inherently complex structure, nearly all chaos-based hash algorithms are less effective than other hash functions.

- **CONSTRUCTION OF HASH FUNCTIONS FROM CELLULAR AUTOMATA**

Dynamical systems called cellular automata (CA) have discrete representations of space and time. A CA is made up of a grid-like array of cells, each of which can be in one of a finite number of potential states. These cells are updated synchronously in discrete time steps in accordance with a local interaction rule. Each cell can be either a 0 or a 1.The present states of a cell's neighborhood of neighboring cells influence its state in the following time step. Ulam and von Neumann first created cellular automata with the intention of creating self-replicating artificial systems that are also computationally ubiquitous. This was done in order to give a formal framework for studying the behavior of sophisticated, extended systems. The work of Wolfram was crucial in this regard.

- **ATTACK METHODS AGAINST HASH FUNCTIONS**

Collision resistance, preimage resistance, and second preimage resistance are three traditional security requirements for a hash function. An assault is a deliberate attempt to compromise one or more fundamental security principles. A hash function can be deemed weak if any of the three security properties—preimage, second preimage, or collision—can be discovered with an effort lower than 2n, 2n, and 2n / 2 (for an n-bit hash function). Hash function attacks can be divided into two categories. The first of them is a general attack, which targets construction techniques and assumes that the underlying primitive is secure in every way. It mostly takes use of the hash functions' flaws in general. All hash functions are susceptible to generic attacks. Another assault is a shortcut or particular attack, which preys on the underlying primitive's structural flaw. Shortcut attacks are techniques for cryptanalysis for a particular hash algorithm. In this part, we discuss various attack types.

ATTACKS INDEPENDENT OF HASH FUNCTIONS

- **Brute force attack**

Preimages and collision for a hash function can be calculated using a brute force attack technique. All hash functions are susceptible to brute force attacks, regardless of their structure or other operational features. In a brute force preimage attack, the attacker evaluates a regular n-bit hash function h with each potential input message up until he finds the provided value h(M), where M is a message. In a brute force second preimage attack, the adversary evaluates the hash function h with each potential input message M M until he finds the value h(M) for the message M and the specified hash function h. In a brute force collision attack, the attacker looks for

two messages (M, M) such that M M and h(M) h(M) for a given hash function h. It takes an attacker roughly 2n computations for a hash function to find the first and second preimages, as well as the Birthday paradox, which is covered next. Due to the Birthday paradox, 2n/ 2 is used for collision.

- **BIRTHDAY ATTACK**

The most pervasive attack against cryptographic hash algorithms is known as the birthday attack. The estimated number of random messages to be evaluated before a preimage or collision is identified with a probability greater than 50% is described by this attack. The birthday paradox is the basis for this attack.

PROPERTIES OF A GENERAL HASH ALGORITHM

Hash function properties are typically chosen based on how they will be implemented. For instance, it is generally advisable to create a hash function that is simple to use. For this, the hash computation must be simple enough for every given message. The hash algorithm must to be able to correctly compress the message data concurrently. By having the aforementioned characteristics, any unwanted intrusion is prevented from replacing or altering the input data without altering its fingerprint or hash value. As a result, anytime two strings are discovered to have the same message digest, we can certainly assert that the strings are identical.

RESEARCH METHODOLOGY

DESIGN OF PROPOSED HASH FUNCTION

Every hash function typically consists of two parts: a construction and a compression function. The construction is the process by which the compression function is being continuously called to process a variable-length message. The compression function is a mapping function that turns a bigger arbitrary-size input to a smaller fixed-size output. Traditionally, key components have not been used in the design of hash functions. However, numerous new attacks have been successfully used against these well-known classical hash functions, like SHA-1, MD5, etc. The majority of recently developed algorithms are based on already established and acceptable designs with a few alterations, as we discussed in the previous part, which states that the security of an algorithm needs to be proven. If the current design makes minimal security claims, the new design will likewise. The design ideas of the well-known MD5 algorithm are likewise the foundation for this method. Furthermore, the proposed technique is more resistant to many of the known MD5 attacks thanks to the inclusion of the key in each round of operation on individual blocks.

There are currently two ways to improve or extend the already existing designs: first, by performing more operations than the existing hashing algorithm specifies (for example, using more rounds of operations instead of four primitive functions in the case of MD5); or by adding more advanced coding or permutation steps (for example, using more scrambling techniques in the case of SHA-1); and second, by increasing the total buffer space and using different mipmaps. The most well-liked and frequently used technique is to construct hash functions using block ciphers as a base. This type of hashing employs a compression algorithm, similar to a block cipher, with two inputs: a block of message and a key. A protocol is currently regarded as strong and secure if it takes at least 2128 operations to attack it. But it is certain that more robust security measures will be required in the near future.

In order to implement the suggested approach, the hash compression function integrates the keyed symmetric key block encryption algorithm into each step or round. Both the sender and the receiver utilize the same key because symmetric key encryption only requires a single key to operate. Key Distribution Center (KDC) is able to share this common key across them over an encrypted channel.

The common session key must be sent to both the sender and the recipient in order for communication to take place. For this, KDC employs the master keys belonging to both parties. No other user on the network may intercept and read the original message or use this session key because only these two parties have their associated private keys. No other user knows the shared secret key except for KDC and both parties involved in the message transmission. As a result, this method aids in confirming the source's identity because the key for sender and recipient is now the same.

The operation of the keyed encryption function and the hash compression function have been combined in this method. Each block's output from the compression function serves as the input for the keyed operation. The output of the compression function is 128 bits long. Additionally, a 64 bit input block is taken at a time by a keyed function. As a result, the output of the compression function is split into two blocks of equal size and length (64 bits each), and is then processed twice: once with the left 64 bits and once with the right 64 bits. The keyed function (encryption function) is then used, one by one, on both 64 bit blocks. The output is 128 bits total, divided into 64 bits for each of the left and right parts. This whole 128 bit output is then used as 128 bit CVq to process the following block of input's compression algorithm.

Assign IV to CV0

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

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 ith 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 (Lth block) as final hash value to the receiver.

Step 13: End

The majority of the hashing algorithms in use today are quick one-way hashing algorithms that offer protection against unauthenticated data modification by an adversary. However, it has been noted that message integrity alone does not provide enough protection against all known attacks. Ordinary error detecting codes, for instance, are insufficient since an adversary can construct the right code after altering the data if the mechanism for generating the code is known. With such algorithms, intentional change cannot be detected. That is, let's say a sender sends a message X along with a calculated hash value h. When the message is interrupted, the intruder retrieves it and modifies this X into X'. He might also obtain a copy of h at the same time, calculate a fresh hash h' for a fresh message X', and send it to the recipient. The receiver at the other end does a new hash calculation using the received version of the message, i.e. X'. The outcome will now be validated one, which is untrue.

However, it is possible to create a cryptographic checksum using straightforward symmetric encryption methods. Use this checksum to guard against illegal data alteration, which could be unintentional or deliberate. Additionally, a hashing system can be made more secure and resistant to attacks if a block cipher encryption algorithm is added. The hash function h is designed in such a way that it is simple to compute h(X) from the message X, but it should be impossible to discover even one message X that will provide this value once one knows h(X). Additionally, it must be impossible to calculate any other message M' that yields the same hash value, i.e., h(X) = h(X'). Any strong block encryption mechanism, whose key is previously known by both the sender and the recipient, may then be given the hash value. The sender will use this key to compute the hash value at its conclusion. The receiver will then reverse the transformation and return the value of h(X) using the same matching key. He applies the function h to the message X that was received at the receiving end before comparing two values of h(X). These two hash values will only appear to be equal if the message is authentic and was not altered after the sender generated the hash.

A new secure hash function's design and implementation primarily involve two fundamental concepts: first, a compression function that can operate on any given input string as a block of data with a fixed length, and second, the use of another function in a cascade fashion so that the output of the compression function can extend the output length up to a string with an arbitrary length. We may sum up our design philosophy for the algorithm as "use tried-and-true methods and create stronger ones." For this, we base our hash function on the compression function and the keyed function, two well-known methods.

- The compression function satisfies the criteria of offering fundamental hash algorithm building blocks.
- Use any keyed function to combine source authentication and message integrity.
- With the premise that it is only coming from the original sender, the suggested technique provides a solution for unauthenticated changes made to the input message that cause the receiver to stop receiving it.

RESULTS AND DISCUSSION

SECURITY OF PROPOSED DESIGN FOR “R-U HASH”

The suggested algorithm is guaranteed to be secure and collision-free because each individual part satisfies its own security requirements. We can offer the following justifications for the algorithm's security:

- Secondary functions f1, f2, f3, and f4 are non-invertible and non-linear in mathematics.

- It follows that the overall bits of f(B2, B3, B4) are also independent of one another if the individual bits of B2, B3, and B4 are independent of one another. It guarantees that the hash function has the desired one-way property.
- The accessing order for input words changes for each cycle of processing.
- To prevent fixed point attacks where the attacker's primary goal is to create a second pre-image or collisions by inserting extra blocks into the input, padding is always done by 1 and then the amount of zeros in the output of the preprocessing phase.
- The output of the step before is used as the input for the following step. As a result, any modifications we make anywhere in any of the blocks will undoubtedly influence the algorithm's final result. Therefore, the output from two different messages will never be the same. It demonstrates a second pre-image resistance, which is a further desirable hash characteristic.
- Basic operations like modular arithmetic, XOR, addition, left shift, right shift, simple permutation, etc. are used by algorithms. As a result, processing doesn't need to take more time.
- Function need not wait for table element or key formation because it is possible to generate the necessary t-table and all 48 bit keys well in advance. Using XOR ensures that output depends on all bits, rather than just nearby ones, which aids in accelerating algorithm performance.

Key Generation and Usage

Both the Dedicated-Key setting and the Integrated-Key setting are used by all widely used keyed hash functions currently in existence. Both employ fixed keys, which means that once a key is dedicated, it will be used for every compression function iteration. However, in the suggested method, we will individually apply two word combinations, Y and Z, to 16 different key combinations in one iteration of the compression function. This method obviously takes longer than a keyless one, and it also adds at least n * 2t to the overhead of computing hashes, where n is the total number of blocks and t is the computation cost for one block, either Y or Z. The computation time will only increase by n * t if we perform it simultaneously in parallel for both of these blocks. The efficiency of hashing now depends on how the key function is implemented. As we just stated, this is a light-weight function that won't require a lot of time or effort to build for the entire message length.

Implementation

On several platforms, the suggested keyed hash function can be efficiently implemented. The program's tables, codes, variables, etc. don't take up a lot of room. We can obtain higher performance and better throughput if we

are allowed to use larger cache memory. Since MD5 and DES's well-proven building blocks are also used in this new hash function architecture, we can anticipate similar performance and space properties. However, due to the implementation and use of a key, it offers higher security in comparison to MD5, SHA-0, SHA-1, etc. As a result, it is a more effective message authentication code than MD5 or SHA-1. It may take a little longer, but it is also more secure and less prone to attack.

Performance Analysis of R-U Hash

Any hash function should adhere to the standards outlined in this chapter's and the chapters before it. At the same time, since we are aware that information is no longer necessary if it is not provided within the needed time period, the algorithm's hashing time shouldn't be excessive. As a result, we evaluated the function for inputs of various sizes and discovered that its execution time was not excessive. We then compared it to other widely used hash functions and discovered that the proposed hash function was either taking the same amount of time as them or less. We have used numerous inputs to run and test the program. Even for the identical input, a separate key is created each time, yielding a distinct hash value. we ran this function with various input sizes (equivalent numbers of test cases for input sizes ranging from 1000 bytes to 50000 bytes). It is discovered that the relationship between average execution time and input size.

Table-1: Time taken in seconds for hashing the file of given input size (1000-20,000 Bytes)

Sample No.	Sample Size					
	25000 Bytes	**30000 Bytes**	**35000 Bytes**	**40000 Bytes**	**45000 Bytes**	**50000 Bytes**
1	0.796541201	0.870928712	0.949689038	0.99974807	1.057885076	1.117469352
2	0.807235634	0.868958903	0.941026227	1.001175721	1.057171417	1.121867163
3	0.802225051	0.874140823	0.937519668	1.009684349	1.041272682	1.109476547
4	0.80932048	0.877931611	0.934373957	1.000985123	1.064755779	1.103416197
5	0.808562943	0.867072997	0.945257763	1.006128411	1.055749417	1.102975765
6	0.81847014	0.879596145	0.953985125	0.994965299	1.05112436	1.121372922
7	0.818606414	0.863960255	0.936540677	0.990444176	1.080804727	1.131000306
8	0.809647085	0.868425283	0.921923838	0.999616958	1.038913693	1.113085432

9	0.817465726	0.866870548	0.931137015	1.002703862	1.058542555	1.112133359
10	0.810994749	0.871677543	0.935372887	1.001870144	1.060740026	1.125126722
11	0.816945449	0.891525295	0.933138237	1.001931908	1.042748525	1.113411996
12	0.815483039	0.87701393	0.986801473	0.989039912	1.062434013	1.10022719
13	0.818223038	0.884060366	0.934745689	0.982780597	1.055889085	1.125198541
14	0.813667665	0.866222402	0.943821569	1.008631986	1.054917735	1.104885254
15	0.811468279	0.882963271	0.941723655	0.996459629	1.047721709	1.131061588
16	0.811198541	0.907709933	0.933254803	0.991503937	1.04203429	1.106922171
17	0.813745946	0.887033972	0.924838377	1.011198954	1.072295576	1.109893471
18	0.809550338	0.935912607	0.935002439	1.000259775	1.045856667	1.107799392
19	0.813248516	0.884661523	0.933699602	0.987944622	1.068559238	1.11485606
20	0.819525379	0.881440085	0.943515341	1.027753518	1.04273212	1.133299436
21	0.813432624	0.868711862	0.93149392	1.00757679	1.056605997	1.144456353

TABLE-2 THE AVERAGE TIME TAKEN FOR THESE SAMPLES

Sample Size	Average time taken in seconds
1000 Bytes	0.341636053
5000 Bytes	0.466690688
10000 Bytes	0.587992724
15000 Bytes	0.694276762
20000 Bytes	0.763360959
25000 Bytes	0.81216944
30000 Bytes	0.879848479
35000 Bytes	0.939469586

40000 Bytes	1.000590654
45000 Bytes	1.055178795
50000 Bytes	1.116663582

We tested the method using a large number of inputs, each of which had a fixed size (let's say, 1000 Bytes). Although each time key is different in this scenario, the execution time is nearly identical as seen in table.

Even for the same input, a separate key is created, leading to a new hash value each time. Because of the padding and fixed initialization vector rules that this method utilizes. As a result, it is also secure from attacks using fixed points and a second pre-image collision. However, there is no way to recover the original message from the hash value. Additionally, it employs the idea of a key to create a hash value, implying that there is no chance for an adversary to compute a hash value for a new message and send it to the recipient in order to forge it. This is because we presume that the key is only known by the recipient and sender and is provided to them by a reliable Key Distribution Center (KDC). In order to successfully execute a brute force attack on an n-bit message digest, the attacker needs to complete approximately 2n hash function calculations. Therefore, before an attacker can successfully conduct a brute force pre-image attack and a brute force second pre-image assault on the R-U hash (a 128 bit digest), it needs at least 2128 calculations. However, as it also employs a 64-bit key, its minimum calculations are 2192, which is superior to MD-5, RIPEMD-160, SHA-1, SHA-0, and other algorithms. Similar to that, a brute force collision assault requires 296 computations. However, it is currently not practicable to produce this enormous amount of computational power.

CONCLUSION

Any two of the following two parameters—memory used for computation or computation time, or time complexity or space complexity—can be used to compare different hash functions. The algorithm's software implementation was evaluated on a system with a Pentium® -4 2.66 GHz Intel-based CPU and 1GB RAM. On the same machine with the identical settings, we also run a handful of the currently used hash functions. We took time into consideration while comparing this technique to other algorithms because a hash function's selection during message transfer is heavily influenced by its low time requirement. Since hashing is like a necessary overhead while communicating for security reasons, there is no standard acceptable value of time regarded for hash functions; nonetheless, it is understood that function should not take longer time for calculation itself. Table

compares the amount of time needed to calculate a hash using different hash functions to the R-U hash. Functions for 1 MB, 5 MB, and 10 MB data files were tested. After MD5 and RIPEMD-128, it demonstrates that the method is the third fastest output. Because it uses a key in the algorithm and thus offers more security than MD5 and RIPEMD-128, we can also make the case that it is slower than those algorithms. The proposed hash algorithm thus offers security, computing speed, message integrity, and source authentication, and it has shown to be a useful and practical function in the field of network security.

REFERENCES

1. NIST, Digital Signature Standard (DSS), Federal Information Processing Standards 186-2, Jan. 2000.
2. NIST, NIST Special Publication 800-90, Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), Mar. 2007.
3. Wong P. and N. Memon N., “Secret and public key image watermarking schemes for image authentication and ownership verification”, IEEE Transactions on Image Processing, vol. 10, no. 10, pp. 1593-1601, 2001.
4. Damgård I., “A Design Principle for Hash Functions”, Crypto '89, LNCS, vol. 435, pp. 416- 427, 1989.
5. Sarkar P., Scellenberg P., “A Parallel Algorithm for Extending Cryptographic Hash Functions”, Indocrypt '01, LNCS, vol. 2247, pp. 40-49, 2001.
6. Carter J., Wegman M., “Universal Classes of Hash Functions”, STOC '77, pp. 106-112, 1977.
7. Naor M., Yung M., “Universal One-way Hash Functions and their Cryptographic Applications”, STOC '89, pp. 33-43, 1989.
8. Bellare M., Phillip Rogaway P., “Collision-resistant Hashing: Towards Making UOWHF's Practical”, Crypto '97, LNCS, vol. 1294, pp. 470-484, 1997.
9. Bertoni G., Daemen J., Peeters M., Van Assche G., “Sponge Functions”, Ecrypt Hash Workshop 2007, May 2007.
10. Guo J., Peyrin T., Poschmann A., “The PHOTON Family of Lightweight Hash Functions”, Crypto’11, LNCS, vol. 6841, pp. 222-239, 2011.

11. Lucks S., “A Failure-Friendly Design Principle for Hash Functions”, Asiacrypt '05, LNCS, vol. 3788, pp. 474-494, 2005.

12. Yasuda K., “A Double-Piped Mode of Operation for MACs, PRFs and PROs: Security beyond the Birthday Barrier”, Eurocrypt '09, LNCS, vol. 5479, pp. 242-259, 2009.