An Enhanced Non-Cryptographic Hash Function

Vivian Akoto-Adjepong
University of Energy and Natural Resources
Department of Computer Science and Informatics
P. O. Box 214 Sunyani, Ghana

Michael Asante
Kwame Nkrumah University of Science and Technology
Department of Computer Science
Private Mail Bag, KNUST, Kumasi, Ghana

Steve Okyere-Gyamfi
Christian Service University College
Department of Computer Science and Information Technology
P. O. Box 3110 Kumasi, Ghana

ABSTRACT
How to store information for it to be searched and retrieved efficiently is one of the fundamental problems in computer science. There exists sequential search that support operation such as INSERT, DELETE and RETRIVAL in O (n log (n)) expected time in operations. Therefore in many applications where these operations are needed, hashing provides a way to reduce expected time to O (1).There are many different types of hashing algorithms or functions such as cryptographic hash functions, non-cryptographic hash function, checksums and cyclic redundancy checks. Non-cryptographic hash functions (NCHFs) take a string as input and compute an integer output (hash index) representing the position in memory the string is to be stored. The desirable property of a hash function is that the outputs are evenly distributed across the domain of possible outputs, especially for inputs that are similar. Non-cryptographic hash functions have an immense number of applications, ranging from compilers and databases to videogames, computer networks, etc. A suitable hash function and strategy must be used for specific applications. This will help efficient use of memory space and access time.

The most essential features of non-cryptographic hash functions is its % distribution, number of collisions, performance, % avalanche and quality which are the properties of the hash function. Basing on the properties assessed using a test suite; the results clearly demonstrated that: the proposed hash function that was developed had better properties as compared to other hash functions.

General Terms
Algorithms and functions.

Keywords
Non Cryptographic Hash Function, % distribution, number of collisions, performance, % avalanche and quality.

1. INTRODUCTION
One basic problem in computer science is how to efficiently search and retrieve stored information.

There exists sequential search that support INSERTION, DELETION and RETRIVAL operations in expected time of O (n log (n)). Therefore in applications where INSERTION, DELETION and RETRIVAL are needed, using hash algorithms help to minimize expected time to O (1).

Hashing is used to store and retrieve information in databases. This deals with key attributes or properties and make use of each individual character numbers in the data or key. To implement keyed tables, hashing is a recommended technique [1][2].

Algorithm for lists, trees and stacks takes time proportional to the data size, i.e., O (n).

In order to locate and retrieve information, hashing is a recommended scheme because is effective and efficient [18].

A suitable hash function and strategy must be used to solve particular problems or for specific application. This will help efficient use of memory space and reduce access time.

There exist different types of hash algorithms such as non-cryptographic hash algorithms or functions, cryptographic hash algorithms or functions, checksums and cyclic redundancy checks [1][3].

Independent of the inputs of a hash functions, they are optimized to work very well in different scenarios. The criteria for optimization is based on the assertion that, with hash functions, there should be equal probability with the generation of each output and a little change in inputs, must result in a huge change in outputs [17].

The main focus of study is non-cryptographic hash functions.

Non cryptographic hash algorithms or functions (NCHFs) take its input as string and compute an integer output (hash index) which represent the position in memory the string is to be stored. One of the important properties of hash functions is the even distribution of outputs across the space allocated or domain, especially for similar inputs.

NCHF's are functions that are designed not to withstand an attacker's effort unlike cryptographic hash functions which is designed to withstand an attacker's effort but are much slower. Therefore, NCHF's are faster at the expense of it not able to withstand attackers' effort. NCHFs are used in a number of applications, ranging from databases and compilers to videogames, dictionaries, computer networks, hash tables and other data structures involved in most programming languages. Fast lookup that is found in hash tables are used by numerous network applications [16].

Such hash functions as stated above are: Pearson hash, FNV hash, Bob Jenkins hash, murmur hash, city hash, buz hash etc. [1] [3] [4].

In computing, memory usage and return time are very important resources to consider in running an application. This is dependent on the particular hash function one chooses to solve a problem.

2. REVIEW OF LITERATURE
Below is a discussion of some popular non-cryptographic hash functions.

2.1 Bob Jenkins hash function
Jenkins is known to be designing hash functions for table lookup. Bob Jenkins created a multi byte keyed function which is made up of a collection of non-cryptographic hash functions. This function can be used to detect data that are

similar in a database and as checksums.

There exist variants of bob Jenkins hash functions such as Jenkins's one at a time hash, lookup2, lookup3 and spookyHash.

There are three fundamental stages in Bob Jenkins hash function:

- Combining key length and initialization value to set up an initial state.
- Mixing of bits of the keys in 12 byte increments.
- Processing of remaining bytes of the key.

2.1.1 Jenkins's one at a time hash

This hash function is the first variant of Bob Jenkins hash function. It was formally published in 1997. The function has three stages as stated above. The One-at-a-Time hash is a considerably simpler algorithm of his design. It quickly reaches avalanche and performs very well. It has been used in several high level scripting languages for their associative array data type as the hash function. Some few bits are mixed weakly in the input data as compared to bits that made up the output hash. By default, the programming language Perl, uses Bob Jenkins one at a time hash but can be implemented by the use of Sip hash.

2.1.1.1 Lookup2

This function succeeded Bob Jenkins one at a time hash. The **lookup2** function is also known as (My Hash). This function is now obsolete because of the other functions that Jenkins has released. It is used in many applications [5].

Lookup2 is found in the following applications:

- SPIN model checker: this checker is for detecting error. Researchers Dillinger and Manolios in a paper about this program noted that lookup2 hash function is commonly used in implementing bloom filters and hash tables [6].
- Netfilter firewall component of Linux: this has taken the place of a collision sensitive function that existed earlier [7].
- Jenkins hash function was used in solving the kalah game application, instead of a more commonly used Zobrist hashing technique that was used; the speed of Jenkins hash on kalahboards and the rule of kalah game which causes a radical alter of the board when performance is low negates the importance of Zobrist incremental hash function [8].

2.1.1.2 Lookup3

Lookup3 hash takes in input data in 12 byte chunks. This is very useful when the simplicity of the function is not as useful as speed. For large data, improved speed will be very useful but how complex a function is can cause consequences in speed.

The hashlittle function for a given length and initialization value provided, computes a hash of a single key. For each mixing iteration, the function reduces the key length by 12 bytes. A part of the function that is most computationally expensive is the mixing of the bits of a given key. When a key reaches a length less or equal than 12 bytes the remaining bits are mixed within the hash function after it is extracted [9].

2.1.1.3 SpookyHash

In 2011, Bob Jenkins brought into the system a 128 bit hash known as SpookyHash. SpookyHash is faster compared to lookup3. SpookyHash is a non-cryptographic hash function released into the public domain. It produces 128 bit keys for array byte of any length. 64-bit and 32-bit hash values can also be produce at a similar speed.

The Spooky Hash allows a 128 bit seed. It is given a name SpookyHash because it was released on Halloween.

Reasons to use spookyHash

- spookyHash is fast - For keys that are short it is one byte per cycle, this comes with 30 cycles of cost for startup. For long keys, it is three bytes per cycle, this occupies only one core.
- spookyHash is good - avalanche is achieved for one(1)-bit and two(2)-bit inputs. It is designed to work for any kind or type of key that is made to be like list of arrays of bytes or array of bytes.

When not to use spookyHash

- When there involve an attacker: The reason is that, spooky Hash is not cryptographic. When a digest is given, an attacker who is resourceful can create a tool which can give a message that is modified having an equal hash as the message originally sent. Such tools written can be used by an opponent who is not resourceful to perform their actions.
- Another case not to use spooky Hash is that Big-endian machines are not in support of its new implementations. Good result can be produced when run on big endians but the results will differ from the use of little endian machines. By default, machines which do not read unaligned data cannot also run spooky hash.

For keys that are long, the inner loop of the SpookyHash:Mix(),takes in 8 byte input data, performs xor operation, and another XOR operation, rotation, and then addition.

Whereas Spooky::Mix() handles keys that are long well, it needs four repetitions for mixing finally, and contribute to huge starting cost which makes keys that are short costive. Therefore ShortHash helps in producing a shorter key of 128 bit hash which has a small cost of startup, and Spooky::End() help minimize the mixing cost that is final [10].

2.2 Murmur hash function

This is a function that is generally suitable for table lookup [11]. This exist in various variants and was created in 2008 by Austin Appleby.

The name is from operations which is simple in sequence and performs a thorough mixing of the bits of a given value: x *= m; x = rotate_left(x,r), performs multiplication and then rotation. This is repeated for about 15 times by using good values for m and r and the value of x will then pseudo randomize. The use of multiplication and rotation has some small weakness when they are used in creating hash functions. Therefore multiplication, shift and XOR operators are recommended for use.

When comparism was made between MurmurHash and other common hash functions, murmur hash had a good performance in the distribution of keys [12]. The earlier MurmurHash2 produces 32bit or 64bit hash value.

Two variations of Murmur hash generate 64bit values. That is MurmurHash64A, designed for 64bit processors, and MurmurHash64B, designed for 32bit processors [13].

Someone who uses MurmurHash2A saw a little bug in the C++ implementation, which causes the C and C++ variants to give different hashes for keys, which the size does not conform to multiples of four. The bug was fixed and the code was updated.

Other people also saw a bug in MurmurHash2: this was that because, the 4byte code was repeated a lot of times, it had a high collision chance. This problem was not a problem that could be fixed but does not cause much problem. While this bug was under investigation there emerged a new mix function that was an improvement on the earlier function and was published as MurmurHash3. The current version of MurmurHash, MurmurHash3, produces 32bit or 128bit hash value.

Murmurhash3's performance is better than MurmurHash2. There was no repetitive flaw, had variants in 32bits, 64bits and 128bits for x86 and x64 machines. The 128bit x64 variant is much faster (that is over five gigabytes per second on 3 gigahertz core 2).

Murmurhash3 passed all test but failed avalanche [11].

2.3 Buz hash function

Buz hash function produces up to 2^{32} hash values that are different, but this function uses a lot of Pascal initialization code. In some programming languages, the pascal initialization is done in the code itself; therefore, there will be no need for an initialization call.

For Buz hash function as in pkp hash function, added noise is from a random table, but in *pkp hash* function, it is from characters in an array. *Buz hash* uses set of 32bit random aliases for every character bit. Because of this, each bit location have one-half of its aliases having one and the other having zero, and these are saved in a table. During hashing, the random aliases of every character bit is XOR-ed. XOR-ing have the chance of inverting every character bit by 50%.

For the Java buz hash, it works for keys that have short length than 65 key bits: this is because it was designed for such keys. Most programs are not limited much because of this because most programs uses character bits less than 64 [14].

Buz function was improved by Robert Uzgalis. This hash function is effective and efficient but the pascal initialization code makes it a little cumbersome. [15]

3. METHODOLOGY

The proposed hash function requires a common initial value and an offset. It uses bitwise operators such as shift, bitwise AND, bitwise XOR and bitwise OR. All these are mixed up with the individual characters of the word to be hashed.

3.1 Steps for mixing individual characters of a word

- For the mixing of individual characters, the ASCII value of the new character (ANC) is left shift with the hash value from previous mix (HPM) which initially holds the offset, and the result is stored as intermediate result 1 (IR1).
- The initial value is mixed with the HPM using the AND operator and the result is stored as intermediate result 2 (IR2).
- IR1 exclusive-OR (XOR) IR2 and the result is stored as intermediate result 3 (IR3).
- HPM is left shift with the initial value and the result is stored as intermediate result 4 (IR4).
- IR3 is OR-ed with IR4 and the resulting value is stored as intermediate result 5 (IR5).
- ASCII value of the new character (ANC) is XOR-ed with the initial value and the resulting value is stored as intermediate result 6 (IR6).
- IR5 and IR6 are XOR-ed and the resulting value is kept in the HPM.

3.2 Summation

This mixing of individual characters is done until the length of the array is reached, and the results represent the final hash used to determine the memory location where the data is to be saved or stored.

This memory address or location is calculated by finding modulo of the hash value using the hash table size.
The individual characters are mixed well enough to help achieve avalanche, better distribution, reduced collision, quality and better the performance of the hash function.

The following abbreviations are used in the module and flowchart of the proposed hash function:

- HPM: Hash value from previous mix. This initially holds the offset.
- ANC: ASCII value for the next character
- IR: Intermediate representation.
- PICM: Module of Proposed hash function individual character mixing (PICM)
- FHV: Final hash value
- MI: Memory index

Fig. 1 and Fig. 2 shows a module of the proposed hash function individual character mixing (VICM) and a flowchart of how the proposed hash function works respectively.

4. IMPLEMENTATION

The research strategy used was experimental research and the approach was quantitative in nature.

NCHFs were run on the A TEST SUITE using data (keys) to check the various popular non-cryptographic hash functions %distribution, number of collisions, performance, %avalanche and quality.

This test suite uses the separate chaining collision resolution strategy to resolve collisions.

Data was run several (50) times on the test suite and the results were collected for further analysis.

The hash function's (Bob Jenkins, Murmur, Buz and the proposed hash functions) properties were compared and results were recorded for further analysis.

5. ANALYSIS AND DISCUSSION

The analysis and discussion was done by considering how the various hash functions performed when their properties were tested.

5.1 Percentage Distribution

When the hash table is well distributed, it helps in efficient use of memory space allocated for the hash table. As proportion of areas that are unused in the hash table increases, it does not reduce search cost. This results in wasting memory.

This means that, the proposed hash function efficiently make use of memory space allocated to the hash table with an average % distribution of 42.3%, followed by Bob Jenkins of 41.0%, 40.3% for Buz and 34.9% hash table distribution for Murmur hash function as shown in Fig. 3. To make efficient use of memory space when running applications, the proposed hash will be the best option to use followed by Bob Jenkins, Buz and Murmur hash function.

5.2 Number of Collisions

The total number of operations that is required to resolve collision (i.e. collision resolution strategies) linearly scales to the number of keys mapping to the same slot or bucket. More collisions results in degrading the performance or the efficiency of the hash function significantly. Non cryptographic hash functions deals with operations such as insertion, find or look up, delete and update of data. When there are much collisions, there will be much time involve in performing these operations which will surely degrade the efficiency and effectiveness of the hash function and will result in increased return time and wasting memory space.

The smaller the number of collisions a hash function generates, the faster and more efficient it is. When there is less number of collisions, it will help result in efficient use of memory space and also contribute to reduced return time when operations such as saving, updating, finding and deleting are performed. This is because the time needed to resolve collision using collision resolution strategy will reduce.

On the average, the proposed hash function had the lowest number of collisions of 15, followed by Bob Jenkins with 16 number of collisions, 17 number of collisions for Buz and Murmur hash function recording the highest number of collisions of 22 as shown in Fig. 4. This shows that the proposed hash function will help make efficient use of memory space and reduce return time when used to run applications. This will help increase work output or productivity followed by Bob Jenkins, Buz and then Murmur hash function.

5.3 Performance or speed

Some hash algorithms or functions are cumbersome ie. Computationally expensive, the amount of time (and, in some cases, memory) taken to compute the hash may be burdensome. Speed is measured objectively by using number of lines of code and CPU benchmark. Also, when there are a lot of collisions and operations are performed in that particular location, it can contribute to increase in time spent in performing the operations and this contribute to reduced work output or productivity. But when collisions are less when a particular hash function is used to store data, it helps to reduce return time and this contributes to increase in productivity or work output.

On the average, the proposed hash function had a better performance of 1249ms, followed by Buz hash function with 1289ms, 45132ms for Murmur hash function and 46041ms for Bob Jenkins hash function as shown in Fig. 5.

Therefore when speed is a priority in running applications that uses hash functions, the proposed hash function is the best, followed by Buz, Murmur and then Bob Jenkins hash function.

This is because Bob Jenkins hash function consists of an offset, initial value and a lot of loops for mixing (individual characters of the key). This consists of a lengthy code than any of the hash functions resulting in spending a lot of time to hash a key.

Murmur hash function consist of an offset, initial value and loops for mixing (individual characters of the key) but not as lengthy as Bob Jenkins hash function.

Buz hash function consist of an offset and requires more Pascal initialization code that comes from a randomized table and mixing (individual characters of the key) using bitwise operators which is quite simple.

The proposed hash function consist of an offset, an initial value and mixing (individual characters of the key) using bitwise operators, which is more simpler and will require less time to hash a key.

5.4 Percentage Avalanche

A hash function achieve avalanche if the resulting hash index or value is widely different if a single key bit differs. Percentage avalanche aids distribution of data because keys that are similar will not end up having similar or same hash values. A hash function that have good percentage avalanche distributes hash values in a uniform manner and this will help minimize the number of collisions and fill the hash table more evenly.

Higher percentage avalanche contribute to reduced return time and efficient use of memory space. This is because, it help spread data in memory and time needed to resolve collision and perform operations such as save, update, find and delete will reduce.

For Percentage avalanche, both Buz and Proposed hash function had 100% throughout. This means that, every bit of the key changed the hash value, followed by Bob Jenkins hash function with a percentage of 85.6% and Murmur hash function with 27.9% which is the lowest percentage avalanche for all length of characters as shown in Fig. 6.This means that both Buz and proposed hash function can help reduce return time more than Murmur and Bob Jenkins hash.

5.5 Quality

Actually, a good hash function has quality between 0.95 and 1.05. If the quality is high, it means the function has a degraded performance and is not efficient. If the quality is less, it means the function has a good performance and is more efficient.

The quality of a hash function is based on other properties like number of collision, percentage distribution etc.

On the average, proposed hash function was more quality with a value of 0.99, followed by Bob Jenkins hash function with 1.02, Buz hash function with 1.04 and 1.23 for Murmur hash function as shown in Fig. 7. Based on the values recorded, the proposed hash function can help reduce return time and make efficient use of memory space hence help increase work output.

Table 1. A table of hash functions and the various properties

		Properties of hash functions				
		Percentage distribution (%)	**Number of collisions**	**Performance or speed (ms)**	**Percentage avalanche (%)**	**Quality**
Hash Functions	Bob Jenkins	41.0	16	46041	85.6	1.02
	Murmur	34.9	22	45132	27.9	1.23
	Buz	40.3	17	1289	100	1.04
	Proposed	42.3	15	1249	100	0.99

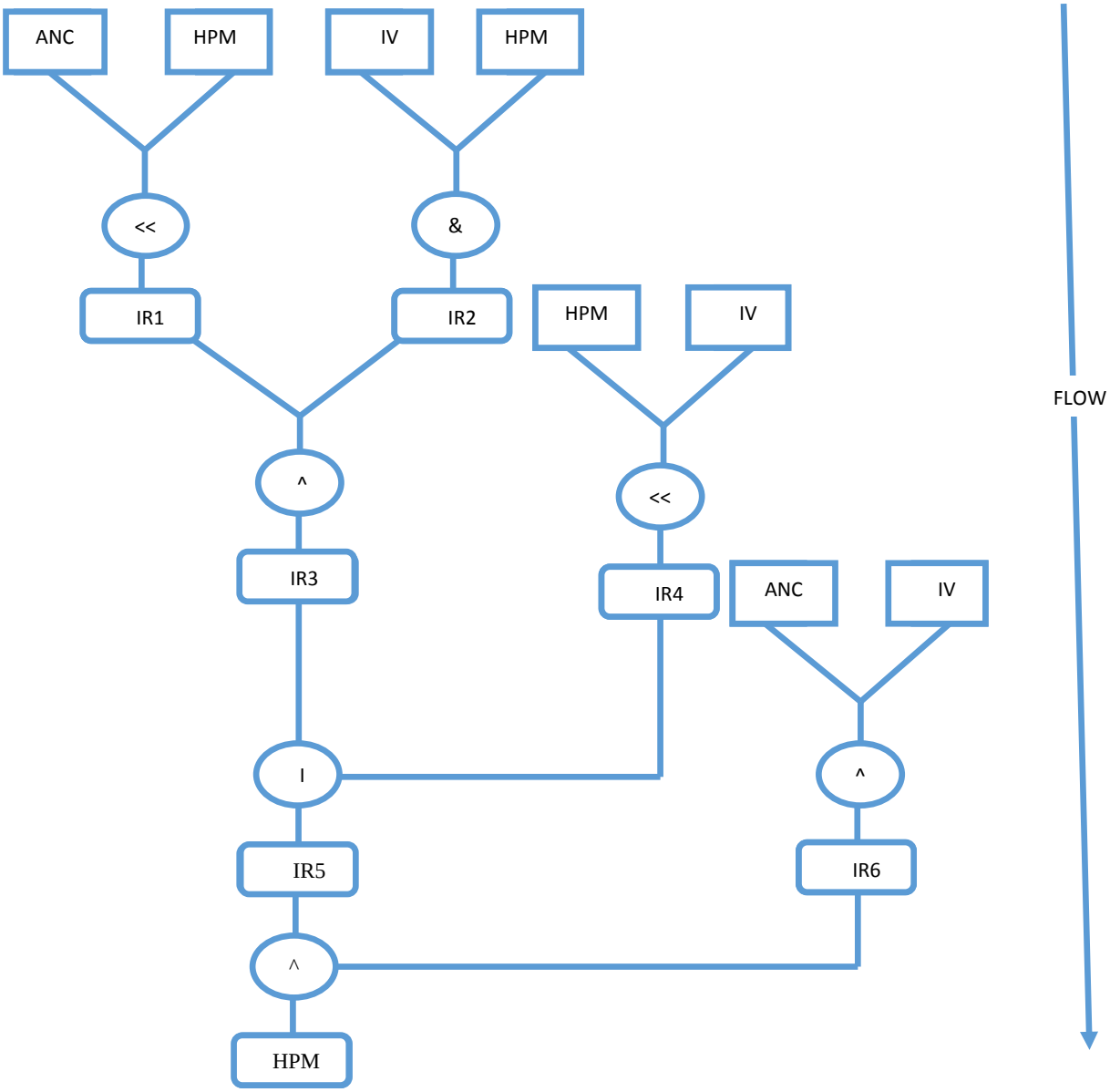

Fig. 1 A module of the proposed hash function individual character mixing (PICM)

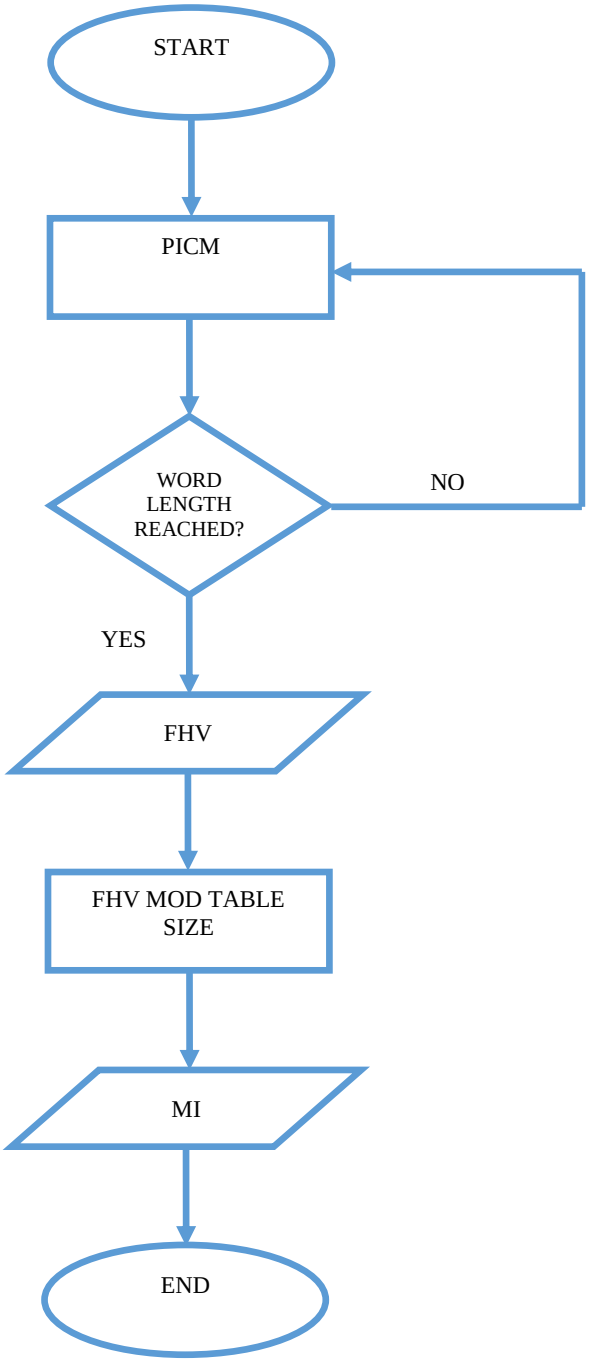

Fig. 2: A flowchart of how the proposed hash function works.

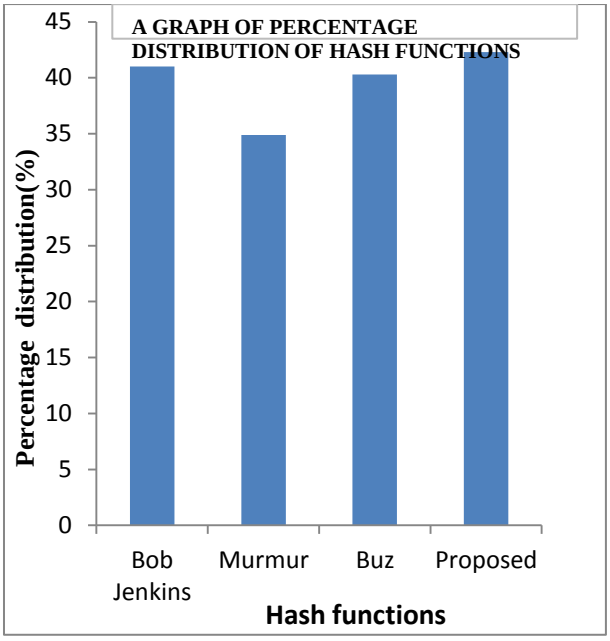

Fig. 3: A graph of percentage distribution of hash functions

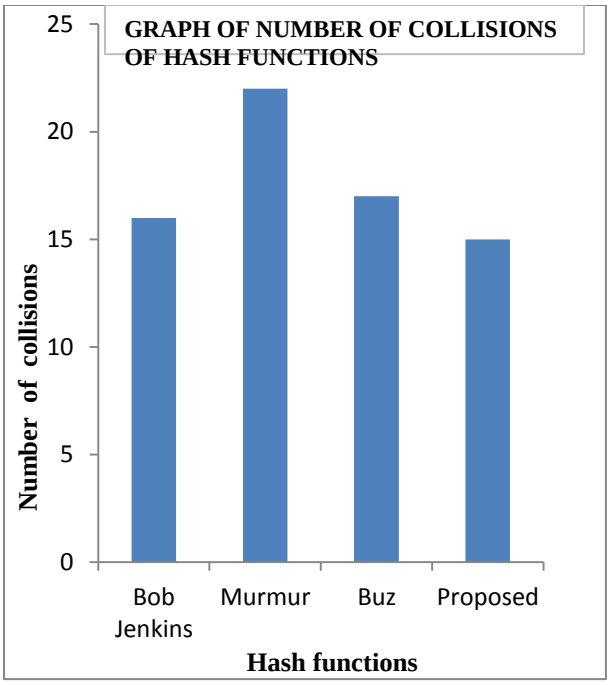

Fig. 4: A graph of number of collisions of hash function

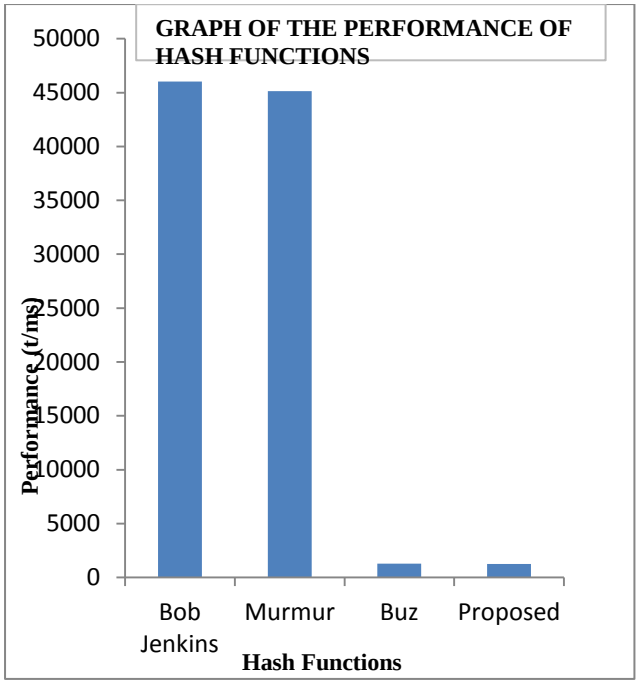

Fig. 5: A graph of the performance of hash functions

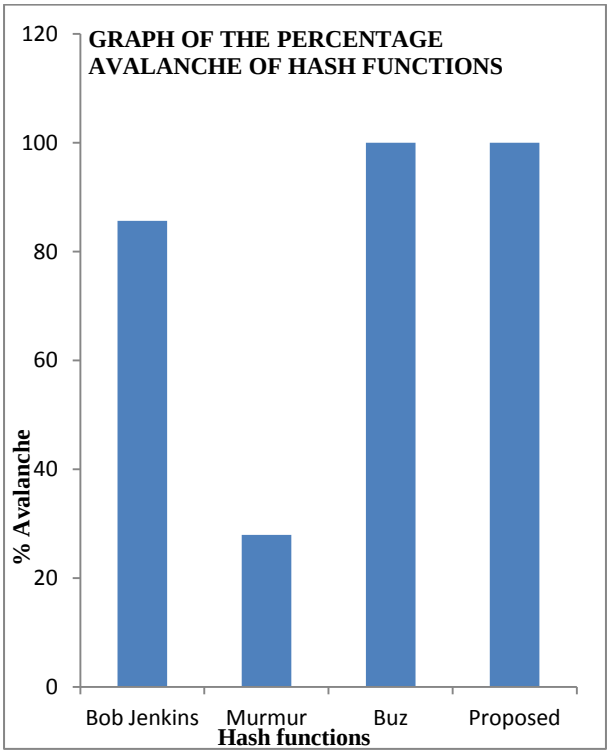

Fig. 6: A graph of the percentage avalanche of hash functions

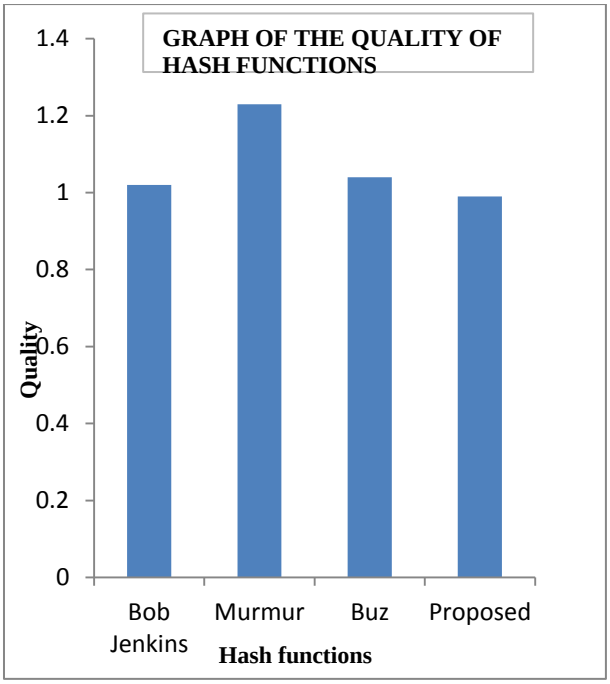

Fig. 7: A graph of the quality of hash functions

6. CONCLUSION

The most essential features of non-cryptographic hash functions is its percentage distribution (which shows how evenly data is spread out in the memory space allocated for the data to be stored or the hash table), number of collisions (which shows the number of keys or data that hashes to the same address space), performance or speed (which shows how fast the hash function can consume input), % avalanche (deals with how each individual key bit, contribute to a change in hash value produced) and quality (which tests the quality of hash function based on the various properties), which are the properties of non-cryptographic hash functions.

Percentage distribution help to make efficient use of memory space when running applications, when data is well distributed in memory space allocated. The proposed hash had the best distribution followed by Bob Jenkins, Buz and Murmur hash function.

For number of collisions which when less for a particular non-cryptographic hash function, it help to make efficient use of memory space and reduce return time, and when used to run applications, contribute to increased work output or productivity, The proposed hash had the lowest followed by Bob Jenkins, Buz and then Murmur hash function.

Also, for performance or speed when less for a particular hash function, it help to reduce return time when used to store data and contribute to increase in productivity or work output. The proposed hash function had a better performance followed by Buz hash function, Murmur and Bob Jenkins hash function. Therefore when speed is a priority in running applications that uses hash functions, the proposed hash function is the best, followed by Buz, Murmur and then Bob Jenkins hash function.

For percentage avalanche which when high contribute to reduced return time and efficient use of memory space, Buz and the proposed hash function had the highest. This means that both Buz and the proposed hash function can help reduce return time more than Murmur and Bob Jenkins hash.

For quality which is dependent on all the other properties. The lower the value, the better the hash function. It depicts how effective and efficient a hash function is in terms of reduced return time and efficient use of memory space, which help to increase work output when used to run applications, the proposed hash function had the lowest value, followed by Bob Jenkins hash function, Buz hash function and then Murmur hash function. This means that the proposed hash function is the most effective.

Based on the properties examined, the results clearly demonstrated that, the proposed hash function had better properties which means that it is more effective and efficient to use to run applications as compared to Bob Jenkins, Murmur and Buz hash functions.

7. ACKNOWLEGDEMENT

Our thanks to the almighty God and all who contributed in diverse ways to make this paper a success.

8. REFERENCES

[1] Singh, M. and Garg, D. 2009. Choosing best hashing strategies, IEEE International advanced computing conference (IACC'09), p 50.

[2] Walker, H. M. 1998. Abstract Data Types, Clarendon Press, 4th Edition: pp. 129-143.

[3] Kumar, C. K. and Suyambulingom, C. Modification on Non Cryptographic Hash Function. International Journal of Computational Engineering Research (IJCER) ISSN: 2250-3005. National Conference on Architecture, Software system and Green computing: p. 29

[4] Zobel, J., Heinz, S. and Williams, H. E. 2001. In-memory hash tables for accumulating text vocabularies. Information processing letters: pp. 271,272.

[5] Bob, J. 1997. Hash functions. Dr. Dobbs Journal.

[6] Dillinger, C. Peter, Manolios, Panagiotis. 2004. Fast and accurate bitstate verification for SPIN. Proc. 11th International SPIN Workshop on Model Checking Software, pp. 57-75.

[7] Ayuso, N. and Pablo. 2006. Netfilter's connection tracking system. (PDF). Login 31(3), pp. 34-38.

[8] Irving, G., Donkers, J. and Uiterwijk J. 2008. 6*6 LOA is solved: kalah. (PDF), ICGA Journal: pp. 234-237.

[9] Lalanne, C., Muralidharan, S. and Lysaght, M. 2015. An OpenCL design of the Bob Jenkins lookup3 hash function using the Xilinx TM SDAccel™ Development Environment. ICHEC White Paper, July 16, pp. 2,3.

[10] Bob, J. 2012. SpookyHash: a 128-bit noncryptographic hash. Retrieved Dec 12, 2017

[11] Couceiro et al." (PDF) (in (Portuguese)). 13 January 2017.

[12] Tanjent. 2008. MurmurHash first announcement. Tanjent.livejournal.com. Retrieved 13 January 2017.

[13] Adam. 2010. MurmurHash2-160. Simonhf.wordpress.com. Retrieved 13 January 2017.

[14] Uzgalis, R.1995. Contact: buz@cs.aukuni.ac.nz

[15] Uzgalis,R.2009."A very efficient java hash algorithm, ba sed on the BuzHash algoritm by RobertUzgalis (http://w ww.serve.net/buz/hash.adt/java.000.html)

[16] Dobai, R. and Korenek, J. 2015. Evolution of Non-Cryptographic Hash Function Pairs for FPGA-Based Network Applications. IEEE Symposium Series on Computational Intelligence, p. 1219.

[17] Estebanez, C., Saez, Y., Recio, G., and Isasi, P. 2014. Automatic design of noncryptographic hash functions using genetic programming. Computational Intelligence, vol. 30, no. 4. doi:10.1002/coin.12033, pp. 798–831.

[18] Coremen, T. H., Leiserson, C. E. and Stein, R. L. "Introduction to Algorithms," 2nd edition, PHI, Chapter 11.