Modification on Non-Cryptographic Hash Function

C. Krishna Kumar[1], Dr. C. Suyambulingom[2]
[1,] Sathyabama University, Chennai, India, [2]Professor (Rtd.), Dept. of Mathematics, TAU, Coimbatore, India

Abstract—
This paper presents an automatic approach to Modification on non-cryptographic hash function design based on grammar guided genetic programming. The paper describes how it is possible to design a non-cryptographic hash function, implementation issues such as terminal and nonterminal symbols, fitness measure, and used context-free grammar. The main aim of this paper is to link the expert knowledge in the design of non-cryptographic hash function and the process of automatic design which can try many more combinations then an expert can. The hash function automatically designed in the paper is competitive with human design and it is compared with the most used non-cryptographic hashes in the field of speed of processing and in the field of collision resistance. The results are discussed in the last section and further improvement is mentioned.

Keywords— Modification on Non-cryptographic Hash Function, Evolutionary optimization, Genetic Programming, Context-free Grammar, Collision Resistance.

I. Introduction

THE hash function is generally understood as a view that assigns an output sequence of always fixed length to an arbitrarily long input sequence, according to a strictly defined algorithm. Formally, hash function is the mathematical function *h* transferring bits of the input sequence *I* to the output sequence *O*, which is of a fixed length. According to Mao [1] the hash functions should have several basic characteristics. The first one is that the function converts any number of input data to output data of always the same length (the output is called hash). The second feature is that a small change of input data during the hashing process will achieve a big change of output data–this effect is called an avalanche effect. The last feature is a high probability that two messages with the same hash value will be identical. The definition of hash function implies the existence of collisions, which means that a pair of input data $(x, y); x \neq y$ have the same hash values $h(x) = h(y)$. Collisions are undesirable, but in the principle they cannot be avoided. Collisions can only be reduced, because if a collision should be completely eliminated, it would be a hash of the same length as the input data, which would be completely a loss of efficiency lying in compression.

In the following text, the term hash function will be understood as a non-cryptographic hash function which is not so complicated as the cryptographic hash function and it is not mentioned to use it in cryptographic applications, but it is widely used in database systems, hash tables, and other data structures involved in most programming languages.

The most common approach for designing a non-cryptographic hash function is the static design by a human expert in the field of cryptography. Hash function design can be a very time-consuming process, because an expert has to try a significant number of hash functions combining various building blocks. And this is the field where the genetic programming can be used with an advantage of automatic process of design, evaluation and interpretation of the result.

The main contribution of this paper is to link the expert knowledge in the design of hash functions and evolution process of genetic programming which can build and evaluate many more combinations of building blocks for a fraction of the time then a human expert. The second goal is to test an evolution framework developed by the authors, mostly used in the field of image filter design, in the other area of interest and to test its competitiveness in the hash function design with the [2, 3]. The rest of the paper is organized as follows. The second chapter summarizes hash function design issues, i.e. determination of function and terminal set, fitness measure, grammar, and implementation parameters. The application of the proposed method and the evaluation of comparison results are described in chapter three. The fifth chapter brings the discussion about reached results. The last chapter summarizes the results obtained in this paper.

Ii. Hash Function Design

The main goal of this paper is to introduce possibilities of an automatic design of non-cryptographic hash function similar to the already known hash functions, such as: APHash [4], BJHash [5], DEKHash [6], DJBHash [7], FNVHash [8], and to demonstrate the competitiveness of design by the evolution framework developed at BUT with the similar research proposed in the articles [2, 3]. The articles [2, 3] propose GPHash by means of genetic programming but completely without any expert knowledge. The proposed hash function in this article is called EFHash and it is implemented by genetic programming and a context-free grammar.

A Set of Terminals

It was much easier to establish a set of terminal symbols, because it involves only three elements. The first element is the previously hashed value (hash value from previous iteration of hash function). The second element is so-called magic number, which is useful e.g. when some function has
two arguments. It is inspired by FNV hash [8], and it brings more non-linearity. The magic number is set to the value 0x811C9DC5 (the value of magic number in decimal system is 2,166,136,231 and it is a prime number of course). The origin of this magic number comes from FNV hash (32bit
FNV hash specifically), where the magic number is used to produce a hash of certain length. The value of magic number is different for all variants of FNV hash. The third value represents character at certain position in the processed input string sequence.

B. Set of Non-Terminals

First, it was necessary to establish a set of functions (nonterminal symbols) which represents basic building blocks for assembling the resulting hash function. This set is composed of such functions which are quickly and easily feasible, because the resulting hash function must be easily applicable to any hardware and has to work very quickly. This set of functions is strongly inspired by existing hash functions.
The set of functions includes operations, such as: right bit rotation (rr), logical exclusive disjunction (xor, ^), logical
disjunction (or, /), logical conjunction (and, &), logical negation (not, ~), summation (sum, +), and multiplication
(mult, *). Attaching both right and left bit rotation or bit shifting can cause introns, and that is why only the right bit rotation is involved..

C. Fitness measure

When designing hash function a major sign of success can be randomness of solution or random distribution of the output bit string, respectively. Entropy, mean value,
correlation coefficient, and others can be used as the fitness function. It does not matter what function is used to measure the performance of the solution, because that does not guarantee an optimal solution. Therefore, it is better to use multiple evaluation functions. In the [2, 3] authors recommend using the function for non-linearity measure between input and output data. There are several definitions of non-linearity and none of them is agreed by most of scientists. If it is decided to use this measurement, a proper function should measure property called the avalanche effect, so-called intersymbol interference.

In our case, the fitness measure is done by a simple measuring of the number of collisions obtained by hashing 106 random strings of 32bit length. First, it is computed hash bit by bit from the 32bit input string by the proposed candidate hash function produced by genetic programming. The 10^6 random strings are hashed by this candidate hash function. Second, it is calculated how many collisions were produced by the candidate hash function, and this is the score of the hash function (lower number is better). The hash function which reaches the minimum number of collisions is declared as the best solution given by the evolution framework.

D. Implementation

TABLE I
Parameters of Genetic Programming Algorithm

Title	Value
Number of generations	500
Size of population	100
Max. depth of individual	20
Crossover rate	80%
Mutation rate	15%
Terminal set	hash, magic_number
Non-terminal set	rr, xor, or, and, not, Sum, mult

When the genetic programming algorithm is designed, several attributes must be set, see Table I. The evolution process was started with the parameters in Table I.

E. Grammar

The aim of grammar is to define the possibilities of connecting of building blocks. Usually, the grammar is defined by an expert with respect to a specific problem
domain. For the design of the non-cryptographic hash function the following grammar has been defined, see Fig. I.

```
Root := E,
E := E ^ E | E or E | E & E | E + E | E * E |
E ^ C | E or C | E & C | E + C | E * C |
E ^ N | E or N | E & N | E + N | E * N |
N ^ C | N or C | N & C | N + C | N * C | F,
F := RR N | RR F | RR E |
NOT N | NOT F | NOT E |N,
C := 0x811C9DC5,
N := hash (from previous iteration, first is
             set to 0) | character (char of
             string in iteration order).
```

Fig. 1. Context-free grammar in BNF notation.

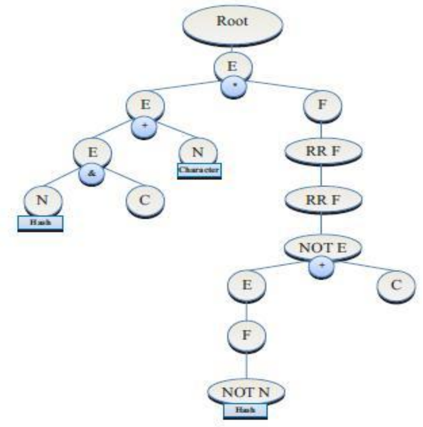

Fig. 2. Resulting hash function as graph of tree based genetic programming

Iii. Results of Experiment

Many different configurations of the terminal and nonterminal sets and fitness function have been tried. The rates of crossover and mutation have been subjected to examination too. And after many weeks of testing the genetic programming algorithm, the appropriate parameters which refer to controlling the run were set, see Table I, and the best solution was finally found.

A. Proposed Hash Function

On the basis of parameters, see Table I, the following candidate solution among all the individuals was found. The resulting hash function is shown in Fig. 2 and Fig. 3.

```
public long run (String str) {
long magic_number = 0x811C9DC5;
long hash = 0;
for (int i = 0; i < str.length(); i++) {
hash  =      ((hash&magic_number)+
              str.charAt(i));
hash=      (hash)*((~((~hash) +
             magic_number)) >> 2);
}
return hash;
}
```

Fig. 3. Resulting hash function algorithm.

B. Hash Functions for Comparison

The following functions were selected as reference functions with which the resulting hash function given by the evolution framework was compared. The best known hash function is FNVHash [8]. The basis of algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee by Glenn Fowler, Phong Vo and Landon C. Noll in 1991. This function is used in domain name servers, databases, web search engines, email servers, antispam filters and many other applications.

DEKHash is an algorithm proposed by the well-known mathematician Donald E. Knuth in *The Art of Computer Programming – Volume 3*, under the topic of sorting and search in chapter 6.4 [6]. DJBHash [7] is an algorithm produced by professor Daniel J. Bernstein and it is one of the most efficient hash functions that has ever been published.

BJHash [5] is a hash algorithm proposed by Robert Jenkins and is widely used in hash tables. APHash [4] is an algorithm produced by Arash Partow, it took ideas from all the above mentioned hash functions to make a hybrid rotative and additive hash algorithm.

GPHash [2, 3] is a hash function produced by the automatic process of genetic programming in 2006. This function is quite fast and the results in the field of collisions are also on a very good level, so it is very good to use it in comparison with the hash proposed by our evolution framework.

C. Speed Test

The speed of processing is one of the methods that can be used for comparison between hash algorithms. This test was carried out as follows. All hash algorithms have been implemented in JAVA programming language, in which all tests were carried out. At the beginning, the strings of 32, 64, 128, 256, 512, and 1,024 bits were randomly generated. In each group there were 106 random strings. The hash function processed each group ten times. These ten measurements were averaged arithmetically and stored. The results of time simulation in seconds are represented in Table II. The headers of the columns represent the length of randomly generated strings in bits. Fig. 4 provides results depicted in the graph. Discussion about reached results is in Section V.

TABLE II
Speed Test of Hash Functions

Algorithm	32	64	128	256	512	1024
APHash	0.076	0.114	0.188	0.346	0.665	1.293
BJHash	0.367	0.413	0.606	0.986	1.816	2.871
EFHash	*0.074*	*0.105*	*0.169*	*0.293*	*0.550*	*1.065*
GPHash	0.074	0.111	0.177	0.301	0.575	1.125
DEKHash	**0.056**	**0.074**	0.104	0.183	0.318	0.598
DJBHash	0.057	**0.074**	**0.103**	**0.181**	**0.314**	**0.59**

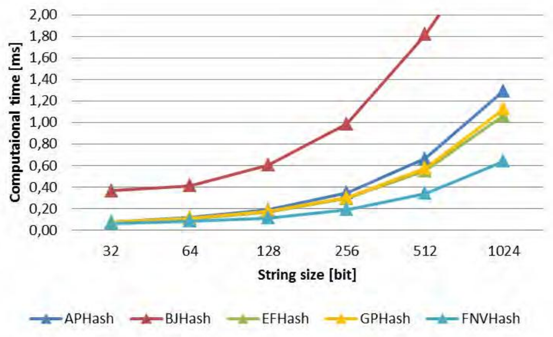

Fig. 4. Speed test of hash functions.

D. Collision Test

This test includes two collision sub-tests. Sub-tests aim to demonstrate the number of collisions obtained by each of the tested hash functions. The first collision test is performed on the same randomly generated strings as the speed test. There are six sets of different bit lengths and each of sets contains unique strings. Strings are randomly generated from characters a-z and 0-9. All sets are mixed up to one set with 106 randomly generated strings in range from 32 bits to 1,024 bits. Hash functions were started on this set. The first 10^2, 10^3, 10^4, 10^5, and 10^6 strings of each set were processed and the results of reached collisions are depicted in Tables III. The results for measure of 10^6 strings were entered to the graph, see Fig. 5. Discussion about reached results is in Section V. The second collision test shows how many hashes (in average) a hash function algorithm can produce before generating the first collision. Besides, it is depicted how the number of collisions growths when the number of hashes growths. The Mersenne Twist Generator was used in this test. The test set consists of random numbers only. Maximum length of random number is 1,024 bits. The hash function was processed ten times. These measurements were averaged arithmetically and rounded up. The results reached are depicted in Table III. The results were also entered to the graphs, see Fig. 6 – 7. This test was processed on 10^7 random numbers and complete number of collision generated by each hash is shown in Table IV and in Fig. 8. Discussion about reached results is in Section V.

Table Iii
Collision Test I

Algorithm	10^2	10^3	10^4	10^5	10^6
APHash	0	0	0	0	**0**
BJHash	0	0	0	2	**119**
EFHash	*0*	*0*	*0*	*0*	*0*
GPHash	0	0	0	0	**0**
DEKHash	0	35	319	1293	**1340**
DJBHash	0	29	288	951	**992**
FNVHash	0	0	0	0	**0**

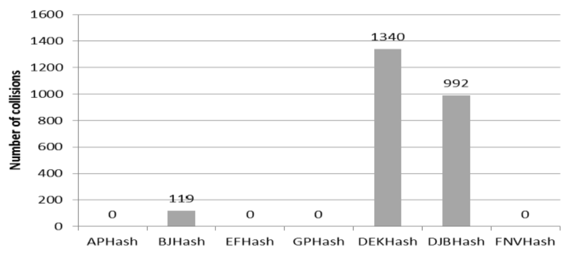

Fig. 5. Collision test I, Number of collisions for 106 strings.

TABLE IV
Collision Test Iia

Algorithm	1	2	3	4	5
Hash	72619	81547	123146	163063	199989
BJHash	117145	168322	178507	182541	188182
EFHash	*73561*	*148137*	*165383*	***251736***	***306792***
GPHash	76519	133368	137280	156540	161076
DEKHash	123094	187204	206570	212548	233598
DJBHash	57724	90071	127812	188884	218495
FNVHash	**152515**	**207282**	**210895**	215018	227487

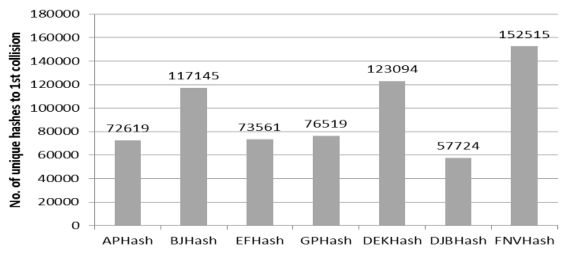

Fig. 6 Collision test IIa, 1st collision.

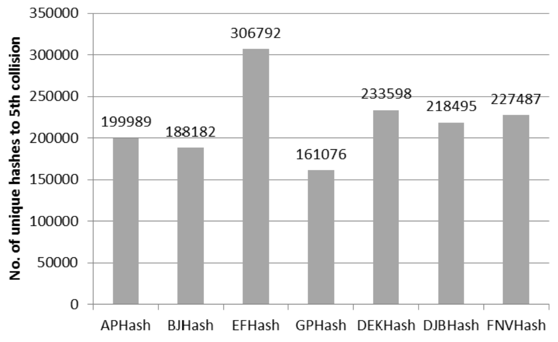

Fig. 7 Collision test IIa, 5th collision.

TABLE V
Collision Test Iib

Algorithm	Total number of collisions
APHash	11697
BJHash	11780
EFHash	11580
GPHash	11708
DEKHash	11766
DJBHash	11525
FNVHash	**11520**

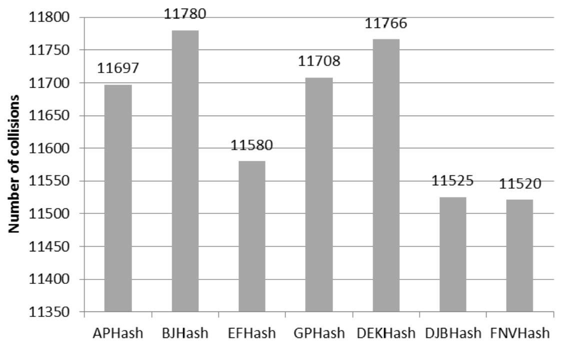

Fig. 8 Collision test IIb.

Iv. Discussion

Based on the results of the first test, it can be concluded that the proposed hash function EFHash is on the average level. However, the results of measurement show that **EFHash can be compared with GPHash**, which is also designed by means of genetic programming. EFHash reached better results than APHash and BJHash, but it is almost twice slower than the fastest DJBHash. The EFHash could be optimized in a speed of processing in the future. The results of measurement the first collision test are shown in Table III. Table III and Fig. 5 shows that EFHash reached very good results on the testing set. There is no collision generated by EFHash during the whole fitness measurement. In contrast, BJHash, DEKHash, and DJBHash generate a significant amount of collisions. The results of measurement of the second collision test are shown in Table IV. Results show that proposed EFHash allows generating the most hashes with only five collisions. On the other hand, the first collision occurred in 73,651 strings, and its suggestion of further improvement. Total number of collisions, which occurred during hashing the set of 107 randomly generated strings by Mersenne Twister Generator are shown in Table V. The resulting values of all algorithms are very similar, but it can be seen that EFHash reached very good position. The EFHash is designed to be a fast and simple hash function with a low collision rate. The EFHash allows to quickly hash lots of data in a various applications. It is possible to make both a software and hardware implementation of this hash function. The proposed hash function can be used in domain name servers, in databases for indexing, in web search or indexing engines, in file systems, in non-cryptographic file fingerprints and so on.

V. Conclusion

This paper introduced an automatic design of Modification on non cryptographic hash function based on the grammar guided genetic programming and the expert knowledge. The paper describes a complete process of genetic programming algorithm design in the field of hash functions. The resulting hash function is introduced and compared to other well known non-cryptographic hash functions. The results obtained for EFHash in both the speed and collision test are shown in the tables and graphs in Section III. All measurements were done in JAVA programming language, and the standard Intel C2D E8400 architecture was used. The non-cryptographic hash function (EFHash) proposed by evolutionary framework could run faster, but in the field of collision avoidance it reaches quite good results in
comparison with other hashing algorithms. Regarding to the competitiveness of design by evolutionary framework, the results in Section IV shows that EFHash can compete with GPHash, because they reached very similar results.

References

[1] C. Estebanez, J. C. Hernandez-Castro, A. Ribagorda, and P. Isasi, "Finding State-of-the-Art Non-cryptographic Hashes with Genetic Programming," in T. P. Runarsson, H.-G. Beyer, E. Burke, J. J. Merelo- Guervos, L. D. Whitley, X. Yao (eds.), PPSN 2005. LCSN, vol. 4193, pp. 818-827. Springer, Heidelberg (2006).

[2] A. Partow, *General Purpose Hash Function Algorithms*, ONLINE http://www.partow.net/programming/hashfunctions/

[3] R. J. Jenkins, *Hash Functions for Hash Table Lookup*, ONLINE, 1995- 1997, http://burtleburtle.net/bob/hash/evahash.html

[4] D. E. Knuth, *The Art of Computer Programming: Volume 3*, Addison- Wesley Professional, 2nd edition, May 4 1998, p. 800, ISBN: 987-0-201-89685-5.

[5] D. J. Bernstein, *Mathematics and ComputerScience*,ONLINE,http://cr.yp.to/djb.html

[6] G. Fowler, L. Noll, P. Vo, *Fowler/Noll/Vo (FNV) Hash*, ONLINE http://isthe.com/chongo/tech/comp/fnv/23

[7] W. Mao, *Modern Cryptography: Theory and Practice*. Prentice Hall PTR, August 4, 2003, p. 648. ISBN 987-0-130-66943-8.

[8] C. Estebanez, J. César, A. Ribagorda, and P. Isasi, "Evolving Hash Functions by Means of Genetic Programming," in *GECCO'06*, July 8-12, 2006, Seattle, Washington, USA