Finding State-of-the-Art Non-cryptographic Hashes with Genetic Programming

César Estébanez, Julio César Hernández-Castro,
Arturo Ribagorda, and Pedro Isasi

Universidad Carlos III de Madrid
Avda. de la Universidad, 30, 28911, Leganés (Madrid). Spain
{cesteban, jcesar, arturo}@inf.uc3m.es, isasi@ia.uc3m.es

Abstract. The design of non-cryptographic hash functions by means of evolutionary computation is a relatively new and unexplored problem. In this paper, we use the Genetic Programming paradigm to evolve collision free and fast hash functions. For achieving robustness against collision we use a fitness function based on a non-linearity concept, producing evolved hashes with a good degree of Avalanche Effect. The other main issue, efficiency, is assured by using only very fast operators (both in hardware and software) and by limiting the number of nodes. Using this approach, we have created a new hash function, which we call *gp-hash*, that is able to outperform a set of five human-generated, widely-used hash functions.

1 Introduction

1.1 Definitions

Hash functions take a message as input and produce an output referred to as a *hash*. More precisely, a hash function h maps bitstrings of arbitrary finite length to strings of fixed length. For a domain D and range R with $h : D \rightarrow R$ and $|D| > |R|$ the function is many-to-one, implying that the existence of collisions (pairs of different inputs with identical outputs) is unavoidable. In the following, the term hash function will refer to non cryptographic hash functions for table and database lookup, mostly used with hash tables [9], not to be confused with the related but quite different cryptographic hash functions usually found in computer security for digital signature and integrity checking. In any case, hash functions should be very efficient (fast) and relatively collision-free (that is, even if we know collisions *should* exist, finding them should be nontrivial).

1.2 A Fitness Function for Hashes

A good way for assuring the quality of a hash function could be to measure the randomness of the hash values produced. There are a number of tests that can be used for this purpose, such as entropy, serial correlation coefficient, average, etc.

One could use any combination of these test as a fitness function for generating highly-random hash functions. However, the common problem to this approach

is that the hash functions obtained need not to pass any other tests than those that form part of the fitness function. Thus, the functions may produce nearly optimal values for all the tests included in the fitness function but quickly fail other, not related, previously unseen tests, even very simple ones.

In this work, however, we propose a completely different approach: instead of measuring output randomness, we measure input/output non-linearity. This change is quite important, because randomness has not a clear definition: it depends on the observer, the tests used, etc. There are multiple definitions for the concept which not satisfy all authors and which, more importantly, make it very difficult, if not impossible, to obtain an undisputed and efficient measure. However, some aspects of non-linearity can be measured by means of a property called Avalanche Effect. In this work, we use this property in the fitness function of a Genetic Programming algorithm for evolving hashes. In this way, we find hash functions that have a very non-linear behavior. Here we show that this generated hash functions can be faster and perform better than other well-known widespread-used hash functions such as FNV Hash [1].

This idea of using evolutionary techniques for generating non-cryptographic hash functions is relatively new: there is only a few works in this topic [7,4,3], and none of them uses a similar approach to ours.

This paper is organized as follows: Section 2 introduces the previously mentioned Avalanche Effect and a stricter variant of it. Section 3 describes our approach and some implementation issues. Section 4 reports the experiments carried out, and the obtained results. Finally, Section 5 draws the main conclusions of the paper.

2 The Avalanche Effect

Nonlinearity can be measured in a number of ways or, what is equivalent, has not a complete unique and satisfactory definition. Fortunately, this is of no concern to us as we do not pretend to measure non-linearity but a very specific mathematical property named avalanche effect because it tries to reflect, to some extend, the intuitive idea of high-nonlinearity: a very small difference in the input producing a high change in the output, thus an avalanche of changes.

Mathematically, $F : 2^m \rightarrow 2^n$ has the avalanche effect if it holds that

$$\forall x, y | H(x, y) = 1, \quad Average\Big(H\big(F(x), F(y)\big)\Big) = \frac{n}{2}$$

So if F is to have the avalanche effect, the Hamming distance between the outputs of a random input vector and one generated by randomly flipping one of the bits should be, on average, $n/2$. That is, a minimum input change (one single bit) produces a maximum output change (half of the bits) on average.

This definition also tries to abstract the more general concept of output independence from the input (and thus our proposal and its applicability to the generation of good hash functions). Although it is clear that this independence is impossible to achieve (a given input vector always produces the same output)

the ideal F will resemble a perfect random function where inputs and outputs are statistically unrelated. Any such F would have perfect avalanche effect, so it is natural to try to obtain such functions by optimizing the amount of avalanche. In fact, we will use an even more demanding property that has been called the Strict Avalanche Criterion [5] which, in particular, implies the Avalanche Effect, and that could be mathematically described as:

$$\forall x, y | H(x, y) = 1, \quad H\big(F(x), F(y)\big) \approx B\left(\frac{1}{2}, n\right)$$

It is interesting to note that this implies the avalanche effect, because the average of a Binomial distribution with parameters $1/2$ and n is $n/2$, and that the amount of proximity of a given distribution to a certain distribution (in this case a $B(1/2, n)$) could be easily measured by means of a chi-square goodness-of-fit test. That is exactly the procedure we will follow.

3 Implementation Issues

We have used the lilgp genetic programming library [2] as the base for our system. Lil-gp provides the core of a GP toolkit so the user only needs to adjust the parameters to fit his particular problem. In this section we detail the changes needed in order to configure our system.

3.1 Function Set

Firstly, we need to define the set of functions: This is critical for our problem, as they are the building blocks of the functions we would obtain. Being efficiency one of the paramount objectives of our approach, it is natural to restrict the set of functions to include only very efficient operations, both easy to implement in hardware and software. Another, but minor, objective was to produce portable algorithms; so the inclusion of the basic binary operations such as **vrotd** (right rotation), **vroti** (left rotation), **xor** (addition mod 2), **or** (bitwise or), **not** (bitwise not), and **and** (bitwise and) are an obvious first step. Other operators as the **sum** (sum mod 2^{32}) are necessary in order to avoid linearity, being itself quite efficient.

The inclusion of the **mult** (multiplication mod 2^{32}) operator was not so easy to decide, because, depending on the particular implementations, the multiplication of two 32 bit values could cost up to fifty times more than an **xor** or an **and** operation (although this could happen in certain architectures, its nearly a worst case: 14 times [6] seems to be a more common value), so it is relatively inefficient, at least when compared with the rest of the operators used. In fact, we did not include it at first, but after extensively experimentation, we conclude that its inclusion was beneficial because, apart from improving non-linearity it at least doubled and sometimes tripled the amount of avalanche we were trying to maximize. That's the reason why we finally introduced it in the function set.

Similarly, after many experiments, we concluded that the functions **vroti** and **vrotd** were absolutely interchangeable and that using them at the same time

was not necessary nor useful, so we arbitrarily decided to remove **vroti** and left **vrotd**. Anyway, with **vrotd** we have a similar problem than with **mult**: compared to other operators, in some architectures **vrotd** is very inefficient so we tried to eliminate this operator and include the $\gg$ (regular right shift) instead. But the problem is that $\gg$ was not able of producing as much non-linearity as **vrotd** and the efficiency gains of the obtained hash functions were not as good as for ignoring the loss of Avalanche Effect.

3.2 Terminal Set

The set of terminals in our case is easy to establish. Firstly, it is mandatory for the hash function to operate with the previous generated hash value. Thus, one of the terminals of the GP system will represent the previous calculated hash value. It will be called **hval**. In our approach, the length of the output v is fixed to 32 bits, so **hval** will be a 32 bits unsigned integer value.

The bitlength of the input (the m value), however, is not that easy to set. Initially, we tried different approaches which did not generated good results, specially in terms of efficiency, so we finally set the input length to 32 bits. In this case, input-related terminals were reduced to a single 32-bit unsigned long value, **a0**. Some experiments confirmed that, as expected, the best obtained 128-to-32-bits hashes were never able to outperform the best 32-to-32 hashes. The later were more efficient, and they usually reached a much higher level of Avalanche Effect.

Finally, we included Ephemeral Random Constants (ERC's) [10] for completing the terminal set. In our problem, ERC's are 32-bits random-values that can be included in the hash function as constants to operate with. The idea behind this operator was to provide a constant value that, independently from the input, could be used by the operators of the function to increase non-linearity, and idea suggested by [12].

3.3 Fitness Function

The fitness of every individual is calculated as follows: First, we use the Mersenne Twister generator [11] to generate two 32-bit random values. Those values are assigned to **hval** and **a0**[1]. As we already know, each individual represents a candidate hash function, so we run the hash function being evaluated with the randomly generated values of **a0** and **hval**. The hash value produced (we call it $hash_1$) is stored. Then, we randomly flip one single bit of one of the two input values, **a0** or **hval**, and we run again the hash function, obtaining a new hash value ($hash_2$). Now, we compute the Hamming distance between $hash_1$ and $hash_2$. This process is repeated a number of times (8192 was experimentally proved to be enough) and each time a Hamming distance among 0 and 32 is obtained and stored. For a perfect Avalanche Effect, the distribution of this

[1] This stands for the 32-to-32-bits hashing. For other input sizes, we only need to use additional **a*** input values.

Hamming distances should adjust to the theoretical Bernoulli probability distribution $B(1/2, 32)$. Therefore, fitness of each individual is calculated by adding two factors: first the measure of how close to 16 ($16/32 = 1/2$) is the mean of the calculated Hamming distances; and second, the chi-square (χ^2) statistic that measures the distance of the observed distribution of the Hamming distances from the theoretical Bernoulli probability distribution $B(1/2, 32)$. Thus, we try to minimize the following fitness expression:

$$Fitness = (16 - mean)^2 + \chi_c^2$$

where χ_c^2 is a corrected value of χ^2, which is calculated as follows:

$$\chi_c^2 = \chi^2 * 10^{-8}$$

where

$$\chi^2 = \sum_{h=0}^{h=32} \frac{(O_h - E_h)^2}{{E_h}^2}$$

and

$$E_k = 8192 * Pr\big(B(1/2, 32) = k\big)$$

We should note that we are computing the value of the χ^2 statistic without the commonly used restriction of adding up only the values when $E_k > 5.0$, for amplifying the effect of a bad output distribution, thus, the sensibility of our measure.

It was necessary to correct the χ^2 statistic because its values were much bigger than the values of the expression $(16 - mean)^2$. Without this correction, the mean measure was negligible and the fitness was guided only by the χ^2.

3.4 Tree Size Limitations

When using genetic programming approaches, it is necessary to put some limits to the depth and to the number of nodes the resulting trees could have. We tried various approaches here, both limiting the depth and not limiting the number of nodes and vice versa. The best results where consistently obtained using this latter option, so we fixed the number of maximum nodes to 25 and did not put a limit (other that the number of nodes itself) to the tree depth. This is also a very important step for assuring the efficiency of the resulting algorithm.

4 Experimentation and Results

The experimentation carried out was extensive. In the GP system part, we tried with many different configurations of the terminal and function sets, the fitness function and the GP parameters, as mentioned in Section 3. Even so, in this

Section we will only show the experiments that produced the most interesting results, in order to save space and do not distract the reader from the important results.

Experiments were carried out in two phases. In the first stage, we use GP to evolve individuals (GP will try to find an individual that minimizes the fitness function described in previous sections). For each configuration and set of parameters described in Section 3 we executed ten GP runs. Using the information provided by the best individuals of each configuration, we selected the parameters that produced better results. This set of parameters is shown in Table 1.

Table 1. Experimentally-found best GP parameters

Parameter	Value
G (Max.Gen.)	2000
M (Pop.Size)	100
Max nodes	25
Terminal and Function set	and or not vrotd xor sum mult a0 hval ERC

Using these parameters, we obtained a large set of candidate individuals. Among them, we selected the best one and called it *gp-hash*. This individual is the best hash function our GP system was able to produce. A description of *gp-hash* can be seen in Figure 1, and its pseudocode in C in Figure 2.

```
(mult 0x6CF575C5
     (vrotd (vrotd (vrotd (vrotd (vrotd
(vrotd (vrotd (vrotd (vrotd (vrotd
(vrotd (vrotd (vrotd (vrotd (vrotd
(vrotd (vrotd (vrotd (mult 0x6CF575C5
      (sum hval a0))))))))))))))))))))
```

Fig. 1. Individual of the generated *gp-hash* function

The second stage starts at this point: We have generated a hash function by means of optimizing the Avalanche Effect, restricting its size and using only the most efficient operators, believing that in this way we would obtain a very fast and relatively collision free hash function. In this stage, we want to check if we have really achieved our objective. In order to do so, we decided to compare *gp-hash* with a set of 5 human-generated non cryptographic hash functions: CRC32, oneAtATimeHash, alphaNumHash, FNVHash [1] and BobJenkinsHash [8]. All of them are state-of-the-art, widely-used hash functions, but within this group FNVHash is well-known to be specially fast and collision free. This justifies its wide adoption in dozens of applications, from NFS implementations (e.g., FreeBSD 4.3, IRIX, Linux (NFS v4)) to Domain Name Servers, not forgetting

```
magic_number = 0x6CF575C5
AUX = magic_number * (hval + a0)
rotate_18_positions_right (AUX)
hash = magic_number * AUX
return hash
```

Fig. 2. C pseudocode of the generated *gp-hash* function

high performance EMail servers, text based referenced resources for video games on the PS2, Gamecube and XBOX, etc.

As the two most important features of a non-cryptographic hash function are its speed and its collision robustness, these will be the two variables that we will test. The former describes how fast the function can hash variable-length bitstrings, and the later is the capability of generating a large amount of hashes while producing as few collisions as possible. So we carried out two different tests: one to compare the speed of the six hash functions, and another one to compare their collision robustness. Both tests were ran in an AMD Athlon XP2000+ with 256 Mb of RAM and a Gentoo Linux Operating System.

4.1 Speed Test

The speed test was designed as follows: All the hash functions are coded in C (none is optimized) and inserted into a speed benchmark. Each run of this benchmark is divided in 32 phases, which we call "executions". In every execution, each function must hash 10^6 random-generated strings. The time took for every function is stored. This process is repeated ten times, and after that, the average time for each function is calculated and stored. Then the execution ends. In the first execution, the length of the random-generated strings is 32 bits. In the second one, this size is multiplied by 2, in the third is multiplied by 3, and so on. Finally, in execution 32, the string size is $32 * 32 = 1024$. This way, when all the executions ends, we have the average time that each hash function needed to hash 10^6 strings of a length varying from 32 to 1024 bits. A summary of these results can be seen in Table 2. Values of the table are average time (in seconds). The headers of the columns are the string size (in bits) of the experiment. Figure 3 shows the graphical representation of the results. It is clear from the results of this experiment that *gp-hash* is faster than the other hash functions, for every string length.

4.2 Collision Test

With the collision test we wanted to know how many hashes (in average) a function can produce before generating the first collision. Furthermore, we also wanted to know how the number of collisions growths when the number of hashes growths. Thus, we created a battery of ten different tests. For each hash function, each test is divided in ten executions. In each execution we store the number of hashes before producing the first collision, and finally we calculate the average

Table 2. Summarized results of the speed test

	32	64	128	256	512	1024
CRC32	0.039	0.068	0.127	0.229	0.435	0.847
oneAtATime	0.059	0.1	0.195	0.367	0.708	1.399
alphaNum	0.033	0.06	0.094	0.222	0.427	0.831
FNVHash	0.039	0.072	0.143	0.267	0.517	1.017
gp-hash	**0.032**	**0.05**	**0.088**	**0.175**	**0.357**	**0.703**
BobJenkins	0.09	0.087	0.164	0.23	0.439	0.82

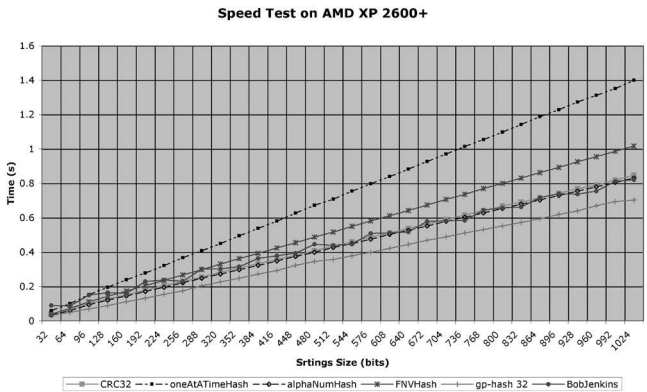

Fig. 3. Results of the speed test

of the ten executions. The second test is similar, but when a first collision is produced, we continue producing hashes and storing values. When a second collision is produced, we store the number of hashes generated. In the third test, we store the number of hashes needed for generate three collisions, and so on.

Results of the complete battery of tests can be seen in Table 3. The chart in Figure 4 shows the way in which the number of collisions growths when the number of hashes also growths. The behavior of all hash functions is almost linear, and it can be seen that *gp-hash* and the other functions have very similar collision-per-hash rates, except oneAtATimeHash which produces significantly worse ratios.

5 Conclusions and Future Work

The results obtained by *gp-hash* in both the speed and collision tests show that this automatically-generated hash function is faster that all the other functions tested when used in the standard AMDXP 2000+ architecture, while its collision rate is absolutely competitive or even slightly better that the rest of the human-designed hash functions, except for one of them (oneAtATimeHash)

Table 3. Summarized results of the collision tests

Hash function	Test 1	Test 2	Test 4	Test 6	Test 8	Test 10
hline CRC32	**323648.38**	541053.83	1033888.5	1445630.91	2033342.26	2583637.81
oneAtATimeHash	71380.66	132346.52	272675.61	407190.86	547254.59	659462.41
alphaNumHash	247355.25	523988.1	1040997.09	**1587157.6**	2079883.36	2657898.19
FNVHash	239840.3	578113.04	**1049437.85**	1559196.5	**2165332.38**	2590886.49
gp-hash	273480.69	**584374.64**	1028586.79	1522586.03	2054192.25	**2670056.92**
BobJenkins	276244.12	554777.21	935246.5	1582715.73	2015192.26	2600531.72

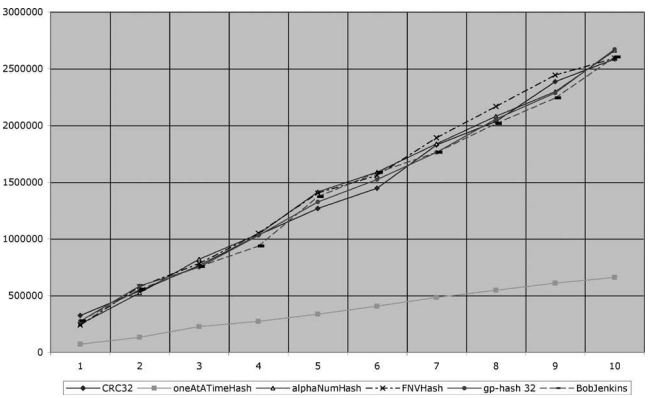

Fig. 4. Results of the collisions test

which performs significantly worse than the rest. So, we can conclude that our proposed system is able to produce competitive hash functions that can outperform other well-known, expert-designed and commonly used hash functions.

Gp-hash, the hash function produced in our experiments and proposed here as an alternative, is slightly faster than FNV Hash (a widespread-used, very fast hash function with many important real-life applications) and adjusts better to the optimal probability distribution $B(1/2, 32)$, or, which is the same, is more non-linear than FNV.

It is important to remark that *gp-hash* was designed in an automatic way. Except for the fitness function, *gp-hash* was generated using no information about the objective, the usage or even the nature of a hash function. Nevertheless, the other hash functions used in the experiments were generated by practiced humans, with years of experience and a vast knowledge about the topic. Even so, *gp-hash* is faster than the rest and able of generating approximately the same number of collisions per hash than the others, in fact winning (the only with FNV who repeats this honor) twice at Table 3. So we have generated an artificial algorithm which can compete on equal terms with those produced by human experts or even beat them.

Acknowledgments

This article has been financed by the Spanish founded research MCyT project OP:LINK, Ref:TIN2005-08818-C04-02.

References

1. Fowler, noll, vo. fnv hash web page, http://www.isthe.com/chongo/tech/comp/fnv/.
2. The lil-gp genetic programming system is available at http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html.
3. P. Berarducci, D. Jordan, D. Martin, and J. Seitzer. GEVOSH: Using grammatical evolution to generate hashing functions. In R. Poli, S. Cagnoni, M. Keijzer, E. Costa, F. Pereira, G. Raidl, S. C. Upton, D. Goldberg, H. Lipson, E. de Jong, J. Koza, H. Suzuki, H. Sawai, I. Parmee, M. Pelikan, K. Sastry, D. Thierens, W. Stolzmann, P. L. Lanzi, S. W. Wilson, M. O'Neill, C. Ryan, T. Yu, J. F. Miller, I. Garibay, G. Holifield, A. S. Wu, T. Riopka, M. M. Meysenburg, A. W. Wright, N. Richter, J. H. Moore, M. D. Ritchie, L. Davis, R. Roy, and M. Jakiela, editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 June 2004.
4. E. Damiani, V. Liberali, and A. G. B. Tettamanzi. Evolutionary design of hashing function circuits using an FPGA, Sept. 17 1998.
5. R. Forré. The strict avalanche criterion: spectral properties of boolean functions and an extended definition. In *CRYPTO '88: Proceedings on Advances in cryptology*, pages 450–468. Springer-Verlag New York, Inc., 1990.
6. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, Q1 2001. http://developer.intel.com/technology/itj/q12001/articles/art 2.htm.
7. D. Hussain and S. Malliaris. Evolutionary techniques applied to hashing: An efficient data retrieval method. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, page 760, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.
8. B. Jenkins. A hash function for hash table lookup. *Dr.Dobbs Journal*, September 1997.
9. D. Knuth. *The Art of Computer Programming*. Addison Wesley, 1998.
10. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
11. Matsumoto and Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8, 1998.
12. D. J. Wheeler and R. M. Needham. TEA, a tiny encryption algorithm. *Lecture Notes in Computer Science*, 1008:363–369, 1995.