Evolving Hash Functions by Means of Genetic Programming

César Estébanez
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911, Leganés (Madrid). Spain
cesteban@inf.uc3m.es

Julio César Hernández-Castro
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911, Leganés (Madrid). Spain
jcesar@inf.uc3m.es

Arturo Ribagorda
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911, Leganés (Madrid). Spain
arturo@inf.uc3m.es

ABSTRACT

The design of hash functions by means of evolutionary computation is a relatively new and unexplored problem. In this work, we use Genetic Programming (GP) to evolve robust and fast hash functions. We use a fitness function based on a non-linearity measure, producing evolved hashes with a good degree of Avalanche Effect. 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.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

Hash functions, genetic programming, avalanche effect

1. INTRODUCTION

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 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).

In this work instead of measuring output randomness, we measure input/output non-linearity. This change is quite important, because randomness has not a clear definition. However, some aspects of non-linearity can be measured by means of a property called Avalanche Effect. Here 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-linearity behavior.

Copyright is held by the author/owner(s).
GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
ACM 1-59593-186-4/06/0007.

1.1 The Avalanche Effect

Avalanche effect 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.

2. IMPLEMENTATION ISSUES

We have used the lilgp genetic programming library [2] as the base for our system. In Table 1 we detail the parameters used to configure our system.

Table 1: Koza Tableau for the hash functions generation problem

Parameter	Value
Terminal Set	ERC, a0 (32 bits input value), hval (previous generated hash value)
Function Set	and, or, not, vrotd, xor, sum, mult
G (max. gens.)	2000
M (pop. size)	1000
Tree size limitations	Max. 25 nodes
Genetic Operators	80% Crossover; 20%Reproduction.

2.1 Fitness function

In the fitness evaluation, we use our hash function with two inputs that are exactly the same except for one single bit which is flipped. When the inputs are hashed, we calculate the Hamming distance between the two generated outputs.

This process is repeated 8192 times, and each time a Hamming distance among 0 and 32 is obtained and stored. For a perfect Avalanche Effect, the distribution of this 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, GP system tries 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: $\chi_c^2 = \chi^2 * 10^{-8}$

3. EXPERIMENTATION AND RESULTS

Experiments were carried out in two phases. In the first stage, we use GP to evolve individuals. Among them, we selected the best one and called it *gp-hash*. Its pseudocode can be seen in Figure 1.

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

Figure 1: C pseudocode of the generated *gp-hash* function.

In the second stage we compare *gp-hash* with a set of 5 human-generated non cryptographic hash functions: CRC32, oneAtATimeHash, alphaNumHash, FNVHash [1] and BobJenkinsHash [3]. All of them are state-of-the-art, widely-used hash functions.

As the two most important features of a non-cryptographic hash function are its speed and its collision robustness, we carried out two different tests: one to compare the speed of the six hash functions, and another one to compare their collision robustness. In the speed test, we calculate the average time that each hash function need to hash $32 * 10^6$ random values of different sizes (between 32 and 1024 bits). In the collision test we measure the average number of hashes that a function can generate before producing the first collision. We also do the same for the second collision, then the third and so on. Results of the speed and collisions tests can be seen in Figure 2 and Figure 3. In Figure 2 y values are hashing times for a set of 10^6 random strings and x values represent the size of the strings. In Figure 3 y values represent the number of hashes generated before a number of collisions is produced; x values are the number of collisions.

4. CONCLUSIONS

With the results shown in Section 3 we can conclude that our GP system is able to produce competitive hash functions that are able to outperform other well-known, expert-designed, commonly-used hash functions.

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. Even so, *gp-hash* is faster and generates approximately the same number of collisions per hash than the others.

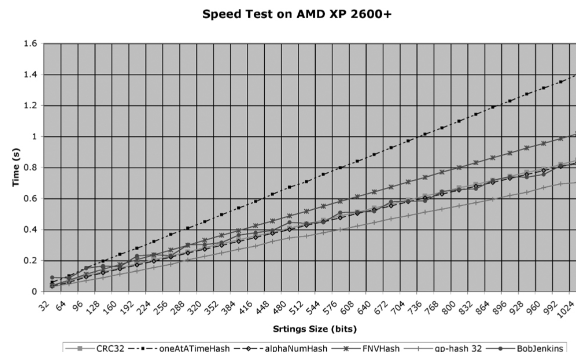

Figure 2: Results of the speed test.

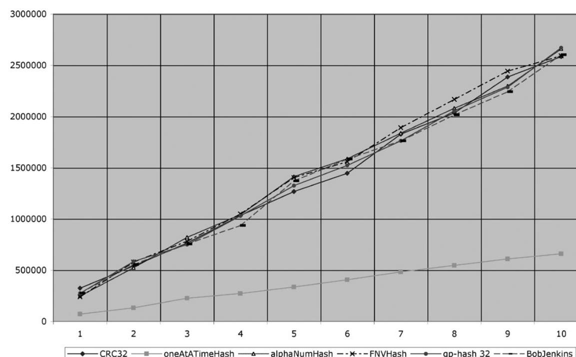

Figure 3: Results of the collisions test.

5. ACKNOWLEDGMENTS

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

6. ADDITIONAL AUTHORS

Additional authors: Pedro Isasi (Universidad Carlos III de Madrid, email: isasi@ia.uc3m.es).

7. 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] B. Jenkins. A hash function for hash table lookup. *Dr.Dobbs Journal*, September 1997.