Automatic Design of Non Cryptographic Hash Functions Using Genetic Programming

César Estébanez, Yago Saez, Gustavo Recio, Pedro Isasi
Department of Computer Science, Universidad Carlos III de Madrid,
Av. Universidad 30 - 28911, Leganes, Madrid, Spain.
email: {cesteban, ysaez, grecio, isasi}@inf.uc3m.es

Abstract

Non Cryptographic Hash Functions have an immense number of important practical applications due to their powerful search properties. However, those properties critically depend on good designs: inappropriately chosen hash functions are a very common source of performance losses. On the other hand, hash functions are difficult to design: they are extremely non linear and counterintuitive, and relationships between the variables are often intricate and obscure. In this work we demonstrate the utility of Genetic Programming and avalanche effect to automatically generate non cryptographic hashes that can compete with state-of-the-art hash functions. We describe the design and implementation of our system, called GP-hash, and its fitness function, based on avalanche properties. Also, we experimentally identify good terminal and function sets and parameters for this task, providing interesting information for future research in this topic. Using GP-hash, we were able to generate two different families of non cryptographic hashes. These hashes are able to compete with a selection of the most important functions of the hashing literature, most of them widely used in the industry, and created by world-class hashing experts with years of experience.

1 Introduction

Hashing is everywhere. Hash functions are the core of hash tables, of course, but they also have a multitude of other applications: Bloom Filters, Distributed Hash Tables, Local Sensitive Hashing, Geometric Hashing, string search algorithms, error detection schemes, transposition tables, cache implementations, and many more. For example, Robert Jenkins reports in his webpage[1] that his hash function *lookup3* has been used by top class companies like Google, Oracle, or Dreamworks (they used it for the Shrek movie). He also reported that *lookup3* was used in implementations of Infoseek, Perl, Ruby, and Linux, among others. The creators of the FNV hash function also report[2] some impressive real-life applications of their function: DNS Servers, NFS implementations

[1] http://burtleburtle.net/bob/other/resume2.html
[2] http://www.isthe.com/chongo/tech/comp/fnv/

(FreeBSD 4.3, IRIX, Linux), videogames (ps2, Gamecube and xbox consoles), Twitter, etc.

Why is hashing so important? The answer is that, under some reasonable assumptions, hashing allows to search for objects in a set in constant time $O(1)$, independently of the size of the set. So, it is not only that the access times are optimal: the most important feature is the perfect scalability of the system. Lookup time remains constant no matter how large the set is. Considering that we live in a world in which governments, companies, and research centers use every day massive databases containing thousands of terabytes of data that must be constantly accessed and updated, it should not be a surprise that hashing is such a popular technique.

Of course, finding elements in time $O(1)$ is the ideal case. In fact, one of the most important drawbacks of hashing is that it has a terrible worst case: finding an object in a set of n elements could has a cost of $O(n)$. This happens only when the hash function maps every input key to the same hash value, and this extreme behaviour is very unlikely as long as we design a decent function. However, performance losses due to unsuitable hash functions are very common. The performance of a hashing system entirely depends on how we design (or choose) the hash function.

1.1 Motivation

The problem is that designing top quality hash functions is a difficult process. They are extremely nonlinear, counterintuitive mathematical constructions in which the relationships between the variables are intentionally obscure and intricate. In fact, most of the non cryptographic hashes that are commonly used in the software industry were handcrafted by experts. Some very popular functions, like FNV, use *magic numbers*, which are numerical constants arbitrarily selected in a trial-and-error process. On top of that, there is no generally accepted way of measuring the quality of non cryptographic hash functions, so, even if one does a good job designing a hash function, it is very difficult to compare it with the state of the art.

These difficulties in the design of good hash functions suggest that Artificial Intelligence (AI) techniques such as Genetic Programming (GP) could do a good job replacing humans in the task of creating new hashes. The reason is that GP is specially suitable for that specific kind of problems: In Poli et al. [2008] authors claim that, based on the experience of numerous researchers over many years, GP is specially productive in problems having some or all of the following properties:

1. The interrelationships among the relevant variables is unknown or poorly understood.
2. Finding the size and shape of the ultimate solution is a major part of the problem.
3. Significant amounts of test data are available in computer-readable form.
4. There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions.

5. Conventional mathematical analysis does not, or cannot, provide analytic solutions.

6. An approximate solution is acceptable.

7. Small improvements in performance are routinely measured (or easily measurable) and highly prized.

We can say that the problem of finding new hash functions completely fulfills at least conditions 1, 3, 4, and 7. And it probably fulfills also all the others in some way.

1.2 Objectives

In this work we want to prove that GP, in conjunction with an adequate fitness function, is able to automatically design NCHF that are competitive with those generated by human experts with years of experience. The great difference with previous works on evolution of hashes is the use of avalanche effect as a powerful estimator of the general quality level of a NCHF. This concept, related to information theory, and widely used in cryptography and hashing, represents the power of the hash to efficiently diffuse input patterns and produce an apparently random output. In this work, we prove that selecting NCHF by their levels of avalanche effect is an efficient, unbiased, and accurate way to discover high quality functions. This allow us to use a very fast mono-objective optimization approach that obtains highly competitive results.

In the next sections we describe the design and implementation of our hashing generation system based on GP. We call this system GP-hash. We also show the experimental work carried out to prove the practical utility of our system, and we claim that GP-hash is able to generate some hashes that compete in performance with state-of-the-art functions that are massively used in industry, like lookup3, FNV, SuperFastHash or MurmurHash.

1.3 Organization

The remainder of this document is organized as follows: In Sections 2 and 3 we introduce respectively NCHF, and GP. Those are the two main technologies on which this work is based. Their sections give a very brief introduction to the most important concepts and suggest further reading to those interested. Then, in Section 4 we review some previous works that involve the application of Evolutionary Computation techniques (and Artificial Intelligence in general) to hashing. Section 5 is dedicated to our GP-hash system: we describe all the design and implementation issues, including fitness function, terminal and function sets, parameter tuning, etc. Then, in Section 6 we use our experimental results to show the utility of GP-hash to generate non cryptographic hashes. Finally in Section 7 we summarize the most important achievements and contributions of this work, and give detailed explanations of what we have learned from it.

2 Non Cryptographic Hash Functions

Hash functions are a family of mathematical expressions that take a message of variable length as input, and return a hash value of fixed length in the output (see Figure 1). This asymmetry between the sizes of inputs and outputs is one of the most important properties of hash functions. Another desirable and important property, also illustrated in Figure 1, is that minimum changes in the input of a hash function should produce maximum changes in the output.

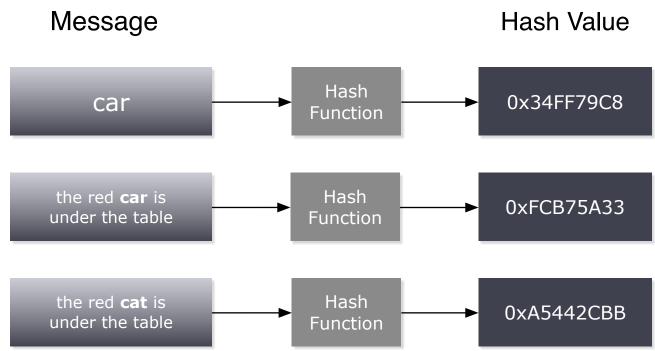


Figure 1: Example of a typical hash function: Input values could have any length; outputs are 32 bits values; The two last inputs only differ in a few letters, but their outputs are completely different.

Most hash functions (both cryptographic and non cryptographic) follow the Merkle-Damgård construction scheme[3] (independently developed by Merkle [Merkle, 1989] and Damgård [Damgård, 1990]). Figure 2 illustrates how it works: inputs of the hash function are split into smaller blocks of fixed size, then blocks are processed one by one by the mixing function, whose mission is to scramble input bits and internal state producing a highly entropic output. In step i, the inputs of the mixing function are block i and the output of processing block $i-1$. If the length of the message is not a multiple of the block size, then a padding must be added to the last block.

There is a huge number of practical applications of hash functions, but the most important one (and the base for most of the others) is the hash table. Hash tables are data structures composed of: a random-access container (e.g. an array) with M slots (usually called buckets) that can store entries; and a hash function. Entries consists of two elements: the data we actually want to store, and a key that identifies the entry. To insert an entry into the table, the hash function is fed with the key, producing a hash value. This value is translated into a valid index of the table, and then the key-data pair is inserted into the bucket indicated by the generated index. When looking for a particular

[3] This does not apply to cryptographic hash functions, which use a variety of systems other than Merkle–Damgård. This is because this construction scheme is no longer considered safe, since different cryptanalysis studies have exposed some weaknesses that are considered important for cryptographic applications. Alternative schema include HAIFA Biham and Dunkelman [2006], *wide-pipe construction* Lucks [2005] and *sponge construction* Bertoni et al. [2007, 2008]

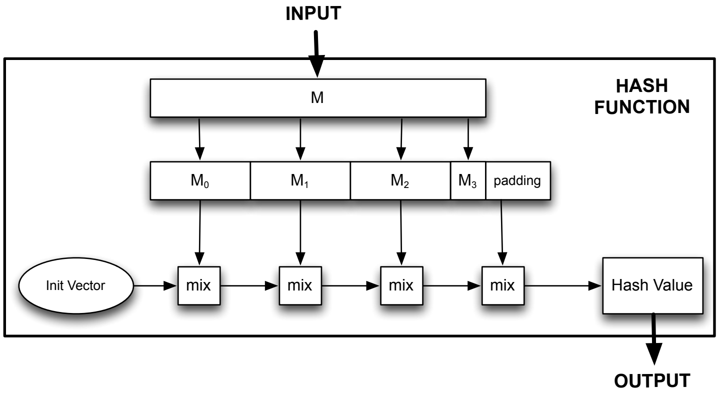


Figure 2: Merkle-Damgård construction scheme.

entry in the table, the process is reversed: the key associated to the entry is hashed and the hash value is translated into an index. The entry is supposed to be in the bucket indicated by the produced index.

Ideally, every hash value should identify a unique input message. However, as stated above, inputs of a hash function have variable size, and outputs have fixed size. This means that there is an infinite number of possible inputs and a finite number of possible outputs. The consequence is that some inputs must produce exactly the same output. We call this a collision. Collisions are an unavoidable problem that can dramatically decrease the performance of hashing.

Apart from collision resistance, we generally require a NCHF to be fast, to distribute outputs evenly, and to produce great levels of avalanche effect.

2.1 Quality criteria for NCHF

According to the hashing literature, the most important quality criteria for NCHF are collision resistance, distribution of outputs, avalanche effect, and speed (see Valloud [2008], Henke et al. [2008], Goodrich and Tamassia [2009]).

- **Collision resistance**: A hash function must reduce the collisions it produces to a minimum [Knott, 1975, Valloud, 2008, Goodrich and Tamassia, 2009]. If we assume that the function produces each hash value with exactly the same probability, it should take about $2^{n/2}$ hash evaluations (where n is the size of the output in bits) to find two colliding keys using a *birthday attack*. However, it could take much fewer if the NCHF is poorly designed [Bellare and Kohno, 2004]. Collisions are one of the major reasons of performance loss in hashing applications, and they should be carefully controlled. Collision resistance is data-dependent: the collision properties of a function can be measured only in relation to a specific key set Knuth [1973], Valloud [2008].

- **Distribution of outputs**: It is very important for a non-cryptographic hash to produce outputs that follow a uniform distribution [Knott, 1975, Sedgewick, 2001, Cormen et al., 2001, Valloud, 2008]. The function must generate each possible output value with the same probability, independent of the distribution of the inputs. An uneven distribution of outputs would produce clustering problems, which greatly affect the performance of a NCHF. Similar to the collision rate, this quality criterion is data-dependent .

- **Avalanche effect**: The avalanche effect of a hash function refers to its ability to produce a large change in output under a minimum change in the input . This property is very important for NCHF [Valloud, 2008, Henke et al., 2008]. A hash with a good avalanche level can *dissipate* the statistical patterns of the inputs into larger structures of the output, thus generating high levels of disorder and preventing clustering problems. This criterion is independent of the architecture and the data, which greatly simplifies its study and measurement.

- **Speed**: NCHF are useful because they allow searches to be performed very quickly. This means that a NCHF must be as fast as possible [Goodrich and Tamassia, 2009, Heileman, 1996, Knuth, 1973, Ramakrishna and Zobel, 1997, Sedgewick, 2001, McKenzie et al., 1990]. For this purpose, NCHF should use very few operators, and these operators should be efficient in terms of CPU consumption. This criterion obviously depends on the architecture in which the hash function runs, since different CPUs offer different performance levels for the same operators (see, for example Matsui and Fukuda [2005]).

2.2 Most common NCHF in the literature

According to their practical applications and their presence in the literature, the most important NCHF are the following:

- **FNV** Fowler et al. [1991]: This function was designed by Glenn Fowler and Phong Vo in 1991, and later improved by Landon Curt Noll. It is one of the most efficient and widely-used hash functions ever created. According to the authors, dozens of very important software products use FNV Hash, including Linux and FreeBSD distributions, Twitter, DNS servers, NFS implementations (FreeBSD 4.3, IRIX, Linux), video games (in PlayStation2, GameCube or xBox consoles), etc. There are two versions of this hash: FNV-1 and FNV-1a.

- **lookup3** Jenkins [1997]: This function was designed by Robert Jenkins and is one of the most important references in the field of non-cryptographic hashes. According to Jenkins, companies such as Google, Oracle and Dreamworks have been using lookup3 in their products. This hash is also included in implementations of PostgreSQL, Linux, Perl, Ruby and Infoseek.

- **SuperFastHash** Hsieh [2004-2008]: This hash was created by Paul Hsieh with the objective of being elegant, extremely fast, and providing high levels of avalanche. It was inspired by some principles found in FNV and lookup3. This function is popular in the software industry: according to Hsieh, Apple uses SuperFastHash in its Open Source project WebKit, which is in turn used in browsers like Safari and Google Chrome. This function was also part of several versions of the former Macromedia product Flash Player.

- **MurmurHash2** Appleby [2008]: This function was designed by Austin Appleby in 2008 and, despite its short lifetime, enjoys great prestige among hashing experts. It is used in some important Open Source projects, like libmemcached, Maatkit, and Apache Hadoop, and has outstanding avalanche properties .

- **DJBX33A**: This function was originally proposed by Prof. Daniel J. Bernstein and is used very often for hashing strings. Many different programming languages such as PHP 5, Python and ASP.NET use DJBX33A or functions derived from it. Java also uses a function that is essentially equivalent to DJBX33A when hashing String objects. This has greatly influenced many application servers, like Tomcat, Geronimo, Jetty or Glassfish, which could be exposed to Denial of Service (DoS) attacks that use known weaknesses of DJBX33A to bring the application server to its knees (see Klink and Wälde [2011], and Crosby and Wallach [2003])

- **BuzHash**: This is a general purpose hash function that was invented by Robert Uzgalis in 1992. It uses a substitution table that replaces each input byte by a randomized alias. These aliases are made so that for every bit position exactly one half of the aliases have a one and the other half have a zero. It is suited for any input distribution, even extremely skewed distributions.

- **DEK**: This is a multiplicative hashing that is based on the ideas of Donald E. Knuth Knuth [1973]. It is one of the oldest and simplest hashing algorithms ever created, and is still very popular in the hashing community. The version used in this work is part of the "General Hash Function Library" by Arash Partow Partow [2010].

- **BKDR**: This function was originally proposed in Kernighan and Ritchie [1988], and is included in the aforementioned "General Hash Function Library".

- **APartow** Partow [2010]: This hybrid rotative and additive hash function algorithm was proposed by Arash Partow and is included in his library of hashes.

2.3 Further Reading

According to Donald E. Knuth, the first publication about hashing is an internal memorandum by H. P. Luhn, an IBM employee, in 1953, but the most cited reference about hashing is Knuth [1998]. It is probably the first textbook that gives a serious introduction to hashing, but its first edition is from the 70's and could be a bit outdated. There are other modern textbooks that also worth the reading: Valloud [2008] is the only textbook that we know which is a comprehensive dedicated guide to hashing. The only con is that the book is focused on SmallTalk. Other textbooks containing interesting chapters about hashing: Sedgewick [2001], Cormen et al. [2001], Goodrich and Tamassia [2009], Heileman [1996]. Another great source of information are the video lectures of the CS course *Introduction to Algorithms* at MIT, publicly available through MIT Open Course Ware.

3 Genetic Programming

GP (Koza [1992]) is a stochastic search technique that tries to automatically generate solutions to a problem starting from high-level statements of what needs to be done. GP belongs to the family of Evolutionary Computation techniques. GP populations are composed of computer programs. Thus, GP part from a random population of programs, and tries to improve them through generations using mechanisms inspired by Natural Selection and Evolution.

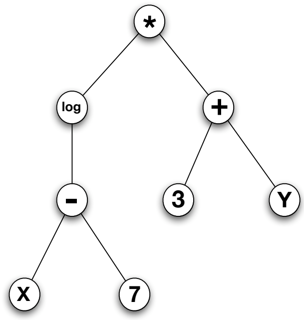
In order to exert a selective pressure over the population and properly guide the search, GP uses a combination of two elements: first, a cost function (or fitness function) that evaluates computer programs and assign them a score indicating their level of adaptation to the problem; and second, a set of operators, that recombine or modify individuals of the population trying to produce fitter programs. As stated in Poli et al. [2008], a typical GP run executes the following algorithm:

1. Randomly create an *initial population* of programs from the available primitives.
2. **repeat:**
3. *Execute* each program and ascertain its fitness.
4. *Select* one or two program(s) from the population with a probability based on fitness to participate in genetic operations.
5. Create new individual program(s) by applying *genetic operations* with specified probabilities (Section 2.4).
6. **until** an acceptable solution is found or some other stopping condition is met (e.g., a maximum number of generations is reached).
7. **return** the best-so-far individual.

Each generation, the current population is evaluated. At the end of this process, each individual has been assigned a numerical value, called the *fitness value*. In some problems, we are looking for low fitness values (e.g. minimize the collisions produced by a hash function), in some others, we want high fitness values (e.g. maximize the disorder produced by a hash function measuring the *randomness* of the outputs). Programs with better-than-average fitnesses are selected to breed and produce new individuals for the next generation. The most common genetic operators used to breed new programs are:

- **Crossover:** The offspring is created by combining randomly chosen parts from two previously selected parents.
- **Mutation:** The new child is created by randomly altering some parts of a previously selected parent.

Individuals are usually represented as parse trees or their equivalent syntactic expressions in polish notation (see Figure 3). Internal nodes of the tree are functions (operators that accept parameters), and leaves are terminals (variables or constants).



(* (log (- X 7)) (+ 3 Y))

Figure 3: Example of GP individual represented as a tree and its equivalent syntax expression.

4 Artificial Intelligence + Hashing

In section 1.1, we explained the reasons why GP could be very suitable to automatically design hash functions: Evolutionary Computation techniques in general are proved to be very good in finding approximate solutions to poorly understood problems, in which the relationships between the variables are not completely known. Furthermore, GP is particularly good at discovering unexpected hash functions because the individuals it evolves does not need to have a fixed size or shape, which is perfect for evolving arbitrary mathematical expressions.

Surprisingly, there is not much research on this topic. Actually, it is hard to find research work that uses AI techniques to automatically generate NCHF.

The most similar work we found is [Safdari, 2009]. In fact, the author cites a previous work that we published in 2006 [Estébanez et al., 2006a] describing the first prototype of our GP-hash system. In his paper, M. Safdari starts with the following family of universal hash functions:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod N$$

And uses a Genetic Algorithm to evolve parameters a and b trying to find the best function for a particular set of inputs. All the databases he uses are sets of random integers lying in a predefined range. The results are promising, but the methodology is questionable: If the input data is purely random, which is the case, then a hash function is not needed at all: just use the input values (or a portion of them) as hash values to obtain a perfect uniform distribution and a minimum collision rate. It will be much more interesting to try the same experiments with biased input sets, which are far more difficult for NCHF.

Furthermore, there are no significance tests (or at least mean values over a number of runs) on the results. The cost function used to guide the search is based in the collision rate and the load factor of the table, two concepts that are mixed in the same function in an apparently arbitrary way.

In a previous work [Berarducci et al., 2004] authors already followed a similar approach, trying to automatically generate hash functions for hashing integer numbers. Their system, called GEVOSH, is even closer to GP-hash than the work of M. Safdari, because GEVOSH uses Grammatical Evolution [Ryan et al., 1998, O'Neill and Ryan, 2003], a technique which is closely related to GP, and because complete hash functions are evolved instead of using a fixed schema and evolve the parameters. The fitness function is based on the collision rate. Two hash functions are obtained and compared with six hashes extracted from [Wang, 2007]. The authors claim that their hashes are competitive with the other six, but the reader can't really tell, because charts in the paper are very difficult to understand (very low quality graphics and no explanation on the text). It is not clear in the paper whether the datasets they used were random or not.

Hussain and Malliaris [2000] is a very short paper in which the authors use a Genetic Algorithm with a collisions-based fitness to evolve some kind of polynomials that they use as hash functions. There is no explanation about how those polynomials are constructed or used to hash, so we assumed that they are using a schema similar to the *Polynomial Hash Codes* studied in [Goodrich and Tamassia, 2009]. The experimental results appear to be good, but the extreme lack of details makes very difficult to evaluate their real impact.

Another interesting flavor of this problem is the automatic design of hashing circuits using Evolvable Hardware. This technique uses evolutionary algorithms to automatically design electronic devices [Sipper et al., 1997, Gordon and Bentley, 2002]. In this domain, [Damiani et al., 1998] offer an interesting approach. They use an evolutionary algorithm to evolve a FPGA-based digital circuit which computes a hash function mapping 16-bit entries into 8-bit hash values. The evolutionary algorithm uses dynamical mutation and uniform crossover. The fitness function is based on the uniformity of the outputs distribution. In [Damiani and Tettamanzi, 1999] this system is adapted to on-line reconfiguration of the circuits. Finally, in [Widiger et al., 2006] we have another example of the application of Evolvable Hardware to the generation of FPGA hashing circuits. In this case, the hash circuits are intended to work as hardware packet classifiers inside routers. The routing rules that the device needs to hash are constantly changing, so the designed hash function must be adaptive, and the circuits must allow on-line reconfigurations. Different hash schemes are used and the results are very interesting.

The automatic generation of cryptographic hashes is completely out of the scope of this work because their design goals and restrictions are completely different from those related to non cryptographic hashes. Even so, we suggest an interesting publication on this topic: [Snasel et al., 2009].

In [Estévez-Tapiador et al., 2008] some of our colleges at Universidad Carlos III de Madrid continued our previous work presented in [Estébanez et al., 2006b] and created a variation of our system that evolves cryptographic hashes. Although we are not dealing with cryptographic functions in this work, we think it is important to cite this work because it shows that the central idea of GP-hash is flexible and powerful enough to be easily adapted to different domains. In the aforementioned paper, authors were able to generate a block cypher that they used as the compression function of a cryptographic hash following the Miyaguchi-Preneel construction scheme [Miyaguchi et al., 1990, Preneel, 1993]. The function they generated was very fast and passed some statistical tests that proves that the function has not evident weaknesses and suggest that it could be secure enough to resist some attacks. They used the same fitness function based on avalanche effect that we developed in our previous work.

4.1 Contributions of GP-hash

In this section we reviewed the previous work on AI applied to the automatic design of NCHF, including Hussain and Malliaris [2000], Safdari [2009], and GEVOSH system proposed in Berarducci et al. [2004]. The most important difference between GP-hash and these systems is the fitness measure: while in previous works collisions rate is always the quality criterion used to evaluate general purpose NCHF, GP-hash uses a fitness function based on the avalanche effect.

This is important because collision rate is data dependent. This means that for calculating the collision resistance of a NCHF, one needs to actually hash a key set and study the frequency of the outputs. So it is only possible to measure the collision properties of a hash with relation to a specific key set. This is a major drawback when trying to evolve general purpose hashes, which should perform well with a wide range of very different key sets.

On the other hand, our approach uses avalanche effect as the main optimization objective. Avalanche is a fundamental characteristic of the internal mixing process of the NCHF, so it does not depend on the hashed key set. This feature makes the avalanche effect a perfect candidate to evolve general purpose hashes. Furthermore, avalanche effect could be seen as a measure of how much disorder the hash function can generate, and how well it disrupts the input patterns. Our hypothesis is that this measure could be a good estimator of the overall quality of a hash function. In Section 6 we experimentally prove this hypothesis, showing how the performance of NCHF evolved with this criteria is comparable to the state of the art in non cryptographic hashing.

5 GP-hash: Design and Implementation

The objective of this work is to automatically discover general purpose, state-of-the-art, NCHF using GP. In order to do so, we used our GP system for automatic generation of non cryptographic hashes. We call this system GP-hash. It is coded in Java, and it makes use of two more publicly available Java libraries: ProGen[4], which provides the GP framework (population management, evaluation and selection, genetic operators, strong typing, etc.), and HashBench[5], which offers a very rich API for NCHF evaluation. A primitive version of GP-hash was previously proposed in [Estébanez et al., 2006b].

[4] ProGen website: http://eva.evannai.inf.uc3m.es/personal/cesteban/cesteban/ProGen.html

[5] HashBench website: http://eva.evannai.inf.uc3m.es/personal/cesteban/cesteban/HashBench.html

In the remainder of this section we explain the decisions made during the design and implementation of GP-hash:

1. Design of the fitness function.
2. Definition of the terminal and function set.
3. Parameter Tuning.

5.1 Fitness Function

In order to design a fitness function for NCHF we considered the quality criteria defined in Section 2:

1. Collisions resistance
2. Distribution of outputs
3. Avalanche effect
4. Speed

The speed is not adequate as the only fitness measure in a mono objective optimization approach: we want our function to be very fast, but that is not enough. This expression for example: **h = 0x0; return h;** is a syntactically valid hash function and it is extremely fast, but it is completely useless. If we use speed as the objective function of a GP run, then we will obtain many individuals like that. The speed could be seen as a secondary objective that have influence on the fitness through a weighted addition (the architecture dependence could be avoided by assigning a cost to each operation proportional to the architecture involved), or it could be considered a constraint of the problem. GP-hash follows the later approach: the size (number of nodes) of the evolved hashes is always limited, so they can only have a limited number of operators. This way, the execution time of the evolved hashes is bounded.

Previous approaches invariably use fitness functions based on collisions resistance properties. As we explained in Section 2.1, collisions resistance is a data dependent metric. This means that it is mandatory to choose a specific key set for training the hash function, which only guarantees that the evolved NCHF will do a good job hashing key sets with similar structures. This lack of generality is a major drawback when evolving general purpose NCHF, which are expected to deliver a proper performance with many key sets of very different natures. The second quality criterion, distribution of outputs, is also data dependent, thus has exactly the same limitation.

One of the most important contributions of GP-hash is the use of avalanche effect as the fitness measure to evolve general purpose NCHF. Avalanche is a intrinsic property of the mixing function of a NCHF, so it is not data dependent. That makes it a perfect candidate to evaluate general purpose NCHF. Furthermore, it is a measure of how well the function disrupts the input patterns and produce an apparently unpredictable output. This feature is intuitively related with a good distribution of outputs (the more random the output looks, the more evenly the outputs distribute) and thus with the collisions rate (biased distributions generate more collisions than pure uniform distributions). This make us hypothesize that avalanche effect could be a very accurate estimator of the global quality of a NCHF.

Although GP-hash also implements fitness functions based on collisions resistance and distribution of outputs, in this work we only use the avalanche effect based fitness. We describe this fitness function next.

5.1.1 Avalanche Fitness

We already introduced the avalanche effect as a quality criterion for NCHF in Section 2.1. In this section, we first provide a more formal definition of avalanche effect and Strict Avalanche Criterion (SAC). Then, we describe the avalanche-based fitness functions designed for GP-hash.

Avalanche effect and SAC: The concept avalanche effect was introduced by Horst Feistel as an important property of *block ciphers* [Feistel, 1973]. Later, this concept was extended to *s-boxes* [Schneier, 1996], cryptographic hash functions [Preneel, 1993], non cryptographic hashes [Valloud, 2008, Mulvey, 2007], etc. We say that a hash function achieves a good avalanche when minimum changes in the input produces maximum changes in the output. This happen if each input bit has some influence on every output bit. The consequence is that flipping a single bit in the input produces an *avalanche* of bit flips in the output. If a hash function achieves high avalanche effect, then the disorder caused by the hash is maximum (see Figure 4).

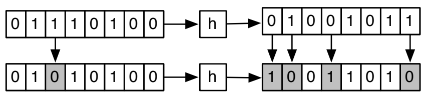


Figure 4: A hash function h with a nice avalanche effect.

A more rigorous concept is the Strict Avalanche Criterion (SAC) introduced in Webster and Tavares [1986]: A hash function satisfies the SAC if for every change in any of the input bits (toggle between 0 and 1) all the bits of the output change with probability 1/2. In other words: flipping one bit of the input changes on average half of the output bits:

$$\forall x, y : \{H(x, y) = 1\} \Rightarrow E\left[H(f(x), f(y))\right] = n/2; \qquad (1)$$

Where $H(x, y)$ is the Hamming distance between x and y; f is a hash function; and n is the number of output bits of f.

Avalanche fitness: A high quality fitness function must deliver smooth and accurate measures, while keeping an eye on efficiency. The SAC is the most precise measurement of avalanche, but checking whether an individual satisfies SAC is not practical: for individuals with 32-bits input and output, that means hashing $32 * 2^{32}$ (i.e. $137,438,953,472$) bitstrings for each individual, for each generation. That is a huge amount of CPU time that we cannot afford. Instead, the avalanche fitness uses a Monte Carlo Simulation: it generates N random bitstrings[6] and the hash values for those bitstrings. Then, for each bitstring, it generates the 32 possible flipped bitstrings (a flipped bitstring is the same original bitstring but with a single bit flipped) and their hash values. Finally, the avalanche function checks the differences between $h(bitstring)$ and each of $h(flippedBitstring)$. Then, there are two possibilities (and two different fitness functions):

1. Measuring the probability $p_{i,j}$ of each input bit i affecting output bit j (i.e. if $p_{i,j}$ is 0.8, that means that if input bit i changes, then output bit j changes 80% of the time). With all the probabilities, construct the avalanche matrix. This matrix contains all the probabilities of every input bit affecting every output bit:

$$AM = \begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,31} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,31} \\ \vdots & \vdots & \vdots & \vdots \\ p_{n,0} & p_{n,1} & \cdots & p_{n,31} \end{pmatrix}$$

   For a perfect avalanche, all probabilities must be 0.5, so we can calculate the total error (we used RMSE) and use this value as the fitness of the individual.

2. Calculate Hamming distances between the hash values of original bitstring and the corresponding flipped bitstrings. We know that those distances should follow a Binomial distribution with parameters 1/2 and n:

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

   This can be used to calculate the goodness of fit using Pearson's Chi Square test:

$$\chi^2 = \sum_{i=1}^{N} \frac{(H_i - n/2)^2}{n/2}$$

   And comparing χ^2 with a chi square distribution of $N - 1$ degrees of freedom we obtain the goodness of fit and the fitness of the evaluated individual.

Both methods work very well, but for the default settings of GP-hash we prefer avalanche matrices because they offer the possibility of nice graphical representations like those shown in Figure 5. The color of square in position (i, j) represents the probability of input bit i affects output bit j. A black square means that changes in bit i do not change bit j at all, or always changes it[7] (0.0 or 1.0 probability of change). A white square means that i has a perfect influence on j (i.e. probability = 0.5).

[6] In our experiments we used $N = 100$ by default.

[7] Note that a probability of 1.0 is as bad as 0.0, because 1.0 means that the value of the output bit is defined by the input bit (every time we change input bit, output bit changes).

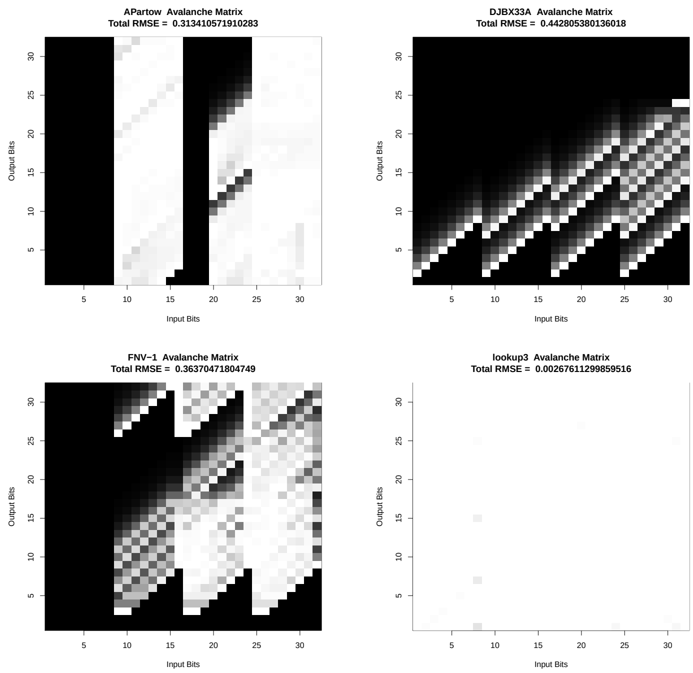


Figure 5: Examples of graphical representation of avalanche matrices corresponding to APartow, DJBX33A, FNV-1, and lookup3 hashes.

5.2 Terminal and Function Set

In this section we explain all the aspects related with the building blocks that GP-hash uses to construct syntactically correct, efficient NCHF. First we give a brief description of the internal representation of the hashing structures. Then, we explain how we choose the terminals and functions of our problem. And finally we show the experimental evidence that support our choice.

5.2.1 Representation of Individuals

The mission of GP-hash is to evolve 32-bits non-cryptographic hashes (we focus on 32-bit NCHF because those are the most common, and we do not want to unnecessarily complicate the explanations, but it is trivial to configure GP-hash to produce functions with different output sizes, e.g. 64, 128, etc.). Given that we already know that virtually all NCHF uses Merkle–Damgård construction, it is an unnecessary waste of time and resources to ask GP-hash to reinvent the wheel: the optimal solution is to provide a Merkle–Damgård construction and make GP-hash to only evolve mixing functions. Internally, individuals of GP-hash only code mixing functions whose inputs in a particular step are the block being processed and the output from the previous step. When we want to externally use those individuals we wrap them with Merkle–Damgård constructions obtaining fully functional hash functions. Mixing functions are coded as regular GP trees.

5.2.2 Terminal Set

Given that evolved functions will follow the Merkle–Damgård scheme, we need at least two different terminals:

- **hval**: 32-bit variable containing the internal state of the hash function. When processing block M_i, *hval* contains the result of processing the previous block M_{i-1}. By default it is initialized to zero, but could be initialized to any other value.
- $\mathbf{a_0}, \mathbf{a_1} \ldots \mathbf{a_n}$: These variables contain the block being processed in the current step. In the Merkle–Damgård scheme, these blocks have a fixed size, but internally the mixing function could process them in separated parts. In the most common case, blocks are 32 or 8 bits long, and we only use one variable $\mathbf{a_0}$ coded in an integer (32 bits) or in a byte (8 bits). However, other combinations are possible and very common in the hashing literature. For example, the mixing function of *lookup3* consumes blocks of 96 bits on each step, and internally the function splits each block in 3 variables of 32 bits length and mixes them separately. To obtain a similar mixing function in GP-hash we should use 3 integer variables $(\mathbf{a_0}, \mathbf{a_1}, \mathbf{a_2})$. By default, we always use one 8-bit variable $\mathbf{a_0}$.

Other important building blocks are *magic constants.* They are very common in hashing literature, in the form of big numbers that are combined with the variables of the system (**hval**, $\mathbf{a_0}$, $\mathbf{a_1} \ldots \mathbf{a_n}$) to improve the overall entropy. There are not established rules about how to choose those numbers, but, in general, prime numbers are preferred, because they are considered to provide more disorder (see for example [Partow, 2010]). In GP-hash, *magic constants* are implemented as *Ephemeral Random Constants* or ERCs (special terminals that are randomly initialized the first time they are evaluated, but that keep their values during the rest of the GP run, as defined in [Koza, 1992]). Each ERC is initialized with an integer randomly selected from a list[8] of one million prime numbers between 15,485,867 and 32,452,843.

$$\text{Terminal Set} = \{\mathbf{hval}, \mathbf{a_0}, \mathbf{a_1} \ldots \mathbf{a_n}, \mathbf{PrimesERC}\}$$

5.2.3 Function Set

The approach we follow to create the function set is to gather some of the most widely used non cryptographic hash functions and check which are the operators that more frequently appear. This way we defined a basic function set by putting together the most common operators in hashing literature. Then we carry out a battery of experiments to refine the basic function set.

In Table 1 we show some of the most important non cryptographic hashes and the operators they use. Addition (+), subtraction (-), multiplication (*) and division (/) are the common arithmetic operators we use everyday[9]. Bitwise operators *xor* (^), *and* (&), *or* (|), and *not* (¬) are also very usual, and do not need explanation. Rigth shift ($\gg$) and right rotation ($\ggg$) are bitwise operators that literally *move* the bits of a variable to the right. The difference between $\gg$ and $\ggg$ is that in the former, bits originally placed in the right end are discarded, and zeros are injected in the left end, while in $\ggg$, bits that are shifted out on the right, are then shifted in on the left. Left shift ($\ll$) and left rotation ($\lll$) operators works exactly the same but in the opposite direction.

Table 1: Operators used by some state-of-the-art non cryptographic hashes.

	+	−	∗	/	shifts	rotations	^	&	|	¬
APartow	√	-	√	-	√	-	√	-	-	√
DJBX33A	√	-	-	-	√	-	-	-	-	-
BKDR	√	-	√	-	-	-	-	-	-	-
lookup3	√	√	-	-	√	√	√	-	-	-
BuzHash	-	-	-	-	-	√	√	-	-	-
FNV	-	-	√	-	-	-	√	-	-	-
MurmurHash2	-	-	√	-	√	-	√	-	-	-
DEK	-	-	√	-	-	-	√	-	-	-
SuperFastHash	√	-	-	-	√	-	√	-	-	-

We observe in Table 1 that almost every non cryptographic hash functions uses a combination of some of the following operators: $\{ +, -, *, \gg, \ll, \ggg, \lll, \hat{}, \neg \}$.

Operators /, &, | are not used by any hash. That is not a surprise given that those operators are not *reversible*. We say that an operator ($\bullet$) is reversible when the operation $x \bullet C = y$ (with C constant) can be reversed, this means, the

[8]This list was obtained from [Caldwell, 1994-2009]

[9]Except for the division, which is protected to avoid divide-by-zero errors and respect the closure property as defined in [Koza, 1992]

value of x can be deduced from the value of y. Using only reversible operators guarantees that the mixing function is reversible, which means that inputs of the function can be calculated out of the outputs. In other words, there is a one-to-one mapping between inputs and outputs, so the mixing function is collision-free. If the function is not reversible, then at least two different inputs must be producing the same output, which means that the mixing function is introducing totally avoidable collisions which will finally propagate to the hash function. See [Mulvey, 2007] for more information about reversible operators and mixing functions.

Multiplication is reversible only in some circumstances and could be slow in some architectures, but it is often used because it introduces a lot of disorder.

Bit shifts are very popular because they are highly entropic and also because they are extremely efficient (only 1 CPU cycle latency on most modern microprocessors). However, they are not reversible unless they are combined with other operators (e.g. **h** $\gg$**= constant** is not reversible, but **h ^= h** $\gg$ **constant** is reversible), so they must be used with care. We cannot expect the GP to be careful when putting building blocks together, so given that bit rotations have a very similar behavior (and efficiency), and that they are always reversible, we tend to prefer rotations in our function set rather than shifts. Furthermore, right rotation and left rotation are completely equivalent (i.e. $(x \ggg n) = (x \lll 32-n)$), so when using rotations we arbitrarily discarded left rotation and keep only right rotation.

Apart from shifts and rotations, the most frequent operators are clearly addition, multiplication and exclusive-or. Thus, we can define a basic function set for GP-hash based on the popularity of the operators:

$$\text{Basic Function Set} = \{+, *, \ggg, \hat{}\,\}$$

5.2.4 Validation of the Terminal and Function Set

Combining the selected functions and terminals we create the basic terminal and function set for GP-hash. Then, following an approach similar to [Wang and Soule, 2004], we carried out a battery of experiments to test whether this set is complete and minimum, and whether our hypothesis about the functions were correct. We include a summary of the results in Table 2. Each row represents the average fitness obtained with different terminal and function sets over 50 runs[10]. Terminal and function sets are labeled as F1,F2 . . . F10. Row labeled as BTFS represents the average fitness obtained with the Basic Terminal and Function Set defined above, and it is used as reference. In the last column there is a symbol that encodes the statistical significance: $\downarrow$ means that results are statistically significant, and $=$ means that there is not significant differences between row average fitness and the BTFS average fitness. It is important to note that we are minimizing fitness values, so the lower the fitness, the better the individual is. We used Shapiro-Wilk test for normality, and t-test and Wilcoxon significance tests for normal, and non-normal distributions respectively (with significance level $\alpha = 0.05$).

Conclusions obtained from the results:

[10] For this experiments we used the avalanche fitness based on avalanche matrices and RMSE explained in Section 5.1.1 and the standard parameters shown in Section 5.3

Table 2: Average results of 50 GP-hash runs with different Terminal and Function Sets. Standard deviation is also shown in parenthesis.

Label	Terminal and Function Set	Avg. Fit. (std dev)	Significance
BTFS	$\{+, *, \ggg, \wedge, hval, a_0, \text{PrimesERC}\}$	0.05026 (0.00084)	n/a
F1	$\{+, *, \ggg, \wedge, \&, \|, hval, a_0, \text{PrimesERC}\}$	0.05134 (0.00167)	=
F2	$\{+, *, \ggg, \wedge, \gg, \ll, hval, a_0, \text{PrimesERC}\}$	0.05206 (0.00550)	=
F3	$\{+, *, \lll, \wedge, hval, a_0, \text{PrimesERC}\}$	0.05015 (0.00112)	=
F4	$\{*, \ggg, \wedge, hval, a_0, \text{PrimesERC}\}$	0.05113 (0.00158)	=
F5	$\{+, \ggg, \wedge, hval, a_0, \text{PrimesERC}\}$	0.23917 (0.00874)	↓
F6	$\{+, *, \ggg, hval, a_0, \text{PrimesERC}\}$	0.0508 (0.00206)	=
F7	$\{+, *, \wedge, hval, a_0, \text{PrimesERC}\}$	0.1739 (0.00060)	↓
F8	$\{+, *, \ggg, \wedge, a_0, \text{PrimesERC}\}$	0.43425 (0.00017)	↓
F9	$\{+, *, \ggg, \wedge, hval, a_0\}$	0.05113 (0.00315)	=
F10	$\{*, \ggg, hval, a_0, \text{PrimesERC}\}$	0.43416 (0.00011)	↓

- **F1** and **F2**: Including & and | does not improve the average fitness of GP-hash. The & operator was never selected for being part of the best individual of a GP-hash run. The | operator was selected only in around 30% of the runs. This is probably related with the non reversibility of those operators, and it is interesting to see that operators which are unpopular among hashing experts are also unpopular in GP-hash solutions. Including bit shifts does not have any effect on the average fitness of GP-hash runs either. Hash functions generated with F2 contains shifts, so shifts are used in the evolution even though they do not improve the performance of just having rotations. These results support our decision of excluding &, | and shifts operators from the BTFS.

- **F3**: As expected, replacing right rotation with left rotation does not have any effect on the average fitness of GP-hash. As we already predicted, both operators are completely equivalent.

- **F4** and **F6**: Surprisingly, removing either addition or *xor* operators from BTFS has no effect on the average fitness. This was completely unexpected: these operators are very popular in the hashing literature, but GP-hash seems to work fine without them. We want to stress that we are talking about two separated experiments: in the first one, we remove addition, in the second one, we remove *xor*. The lack of impact on the fitness could be explained if this two apparently important functions belongs to a *function group* as defined in [Wang and Soule, 2004]. We tested this possibility with function set F10.

- **F5** and **F7**: On the other hand, removing either the multiplication or the rotation, does have a drastic impact on the average fitness. Both changes produces a significant worsening of GP-hash performance. These operators are clearly needed.

- **F8** and **F9**: We also tested *hval* and PrimesERC impact on the average fitness. Results show that the *hval* terminal is definitely needed for a correct evolution. That was totaly expected. What was unexpected is that PirmesERC seems not to be needed. Removing it from the BFTS does not affect the average fitness.

- **F10**: Removing both addition and *xor* operators produces an important worsening of average fitness. As we suspected from the results of F4 and F6, *xor* and addition form a *function group*. In other words, at least one of these operators must appear in the function set, but it does not matter which one. This explains the apparent lack of effect over the fitness of these so popular operators observed in F4 and F6. Is interesting to note that every hash function in Table 1 that does not use addition uses *xor*, and vice versa. According to [Wang and Soule, 2004] the optimal solution is to choose only one of those operators for the function set. Since we already have an arithmetical operator ($*$), but we do not have any boolean operator, we arbitrarily decide to include *xor* and remove addition from the BTFS.

Finally we have defined the terminal and function set for GP-hash:

$$\text{Terminal and Function Set} = \{*, \ggg, ^\wedge, hval, a_0\}$$

5.3 Parameter Tuning

We made an extensive experimental work to find the best parameter set. We followed a similar approach to that of Section 5.2.4: starts with an initial arbitrary configuration based on our knowledge about the problem and our experience working with GP; then, using this basic configuration as a reference, try different changes on the parameters, looking for fitness improvements.

We started our experiments with the basic configuration shown in Table 3, and we progressively introduced changes in all the important parameters: genetic operators rates (±30% to each one), tournament sizes (±5), population size (100, 200, 500 and 1000), initialization method (grow, full, and half and half) , init depth interval (2-4, 2-6, 3-6 and 4-6) and size limitations (25, 50 and 75 nodes). We could not find any configuration which significantly improved the average fitness over the basic tableau. Furthermore, we found out that GP-hash system is very robust: With a large number of different parameter configurations GP-hash keeps working fine, obtaining approximately the same average fitness, and very similar best individuals. Only when using extreme values the average fitness is significantly deteriorated. This is not surprising, since the GP is well known to be a very robust technique in general [Poli et al., 2008, Section 3.4].

We were specially careful in tuning the maximum number of generations: we started from 50 generations and tried rising this parameter. We found out that in GP-hash, evolution curves show very large fitness improvements in earlier generations, and very small improvements later on. This is a very typical behavior of GP populations, as stated on [Luke, 2001]. The improvements obtained with long runs are not proportional to the amount of extra CPU time needed. Therefore we prefer the initial value of 50 generations per run.

The conclusion is that, in the light of our experimental results, we can keep the basic tableau of Table 3 as the default parameters for GP-hash.

Table 3: Basic Tableau for GP-hash.

GP-hash Tableau	
Max Generations	50
Pop. Size	100
Max Nodes	25
Terminal and function set	$\{*, \ggg, \hat{}\,, hval, a0\}$
Fitness	Avalanche Matrices (RMSE)
Crossover	Rate = 0.8 Selection = Tournament Tournament Size = 4
Point Mutation	Rate = 0.1 Selection = Tournament Tournament Size = 4
Reproduction	Rate = 0.1 Selection = Fitness Proportional
Elitism	NO
Initialization	Half and half, init depth 2-4

6 Experimental results

The main hypothesis of this work is to show that evolutionary techniques such as GP can substitute human experts in the challenging task of designing high quality NCHF. In order to prove that, we created GP-hash, a GP based system for the evolution of general purpose NCHF that uses avalanche effect as the global quality estimator for evolved hashes. In this section, we show how GP-hash can be used to generate families of NCHF that are able to compete with a selection of the most widely-used NCHF of the literature. First, we describe the methodology followed to carry on the experiments. Then we present the results, and finally we discuss them.

6.1 Methodology

The experiments we carried out in order to prove the practical utility of GP-hash are divided in two different stages. In the first stage, we use the GP-system described in previous section to evolve a family of NCHF: we use the avalanche effect fitness function, and all the parameters previously specified to perform 50 independent GP-hash runs. This generates 50 NCHF.

In the second stage, we select the best five NCHF produced in previous stage, and compare those functions with a selection of ten of the most widely-used, general-purpose NCHF of the literature: FNV (both versions FNV-1 and FNV-1a), lookup3, SuperFastHash, MurmurHash2, DJBX33A, BuzHash, DEK, BKDR, and APartow (we already described these functions in Section 2.2). The comparison is made in terms of global performance. This mean that we compare our evolved functions with the state of the art in terms of the most important quality criteria for NCHF: avalanche effect, distribution of outputs, and collision resistance (already introduced in detail in Section 2.1). Two of those criteria (collisions resistance and distribution of outputs) are data-dependent. This means that collisions and distribution measurements can only be calculated with relation to a particular key set. Thus, in order to perform reliable comparisons we must compile a collection of key sets that represents the general features of most common hashing problems.

In the remainder of this section, we describe the metrics used to compare NCHF under each criterion, and the key sets we designed for the data-dependent benchmarks.

6.1.1 Metrics

These are the metrics used to compare the performance of each NCHF under the different quality criteria:

- Distribution of outputs: We use Bhattacharyya distance as a measure of how close the outputs of a NCHF are to the ideal uniform distribution. Bhattacharyya distance is a similarity measure that can be used to determine the degree of coincidence of two statistical distributions. It is closely related to χ^2 statistic. In fact, in Aherne et al. [1998], authors deduce that the Bhattacharyya coefficient approximates χ^2 statistic, avoiding in addition some drawbacks that the later is vulnerable to.

  To obtain the Bhattacharyya distance, we calculate first the frequency vector of the NCHF over a key set K, defined as: $X = \{x_0, x_1, \ldots x_{n-1}\}$, where n is the number of possible outputs of the hash function, x_i is the number of times that the hash value h_i was generated, and $p(x_i)$ is the probability of x_i (i.e. $p(x_i) = x_i/|K|$). Then, we calculate the Bhattacharyya distance between the frequency vector and the ideal uniform distribution using Equation 2:

$$D_B(X) = -\ln\left(\sum_{i=1}^{n} \sqrt{p(x_i)\frac{1}{n}}\right) \tag{2}$$

- Collision Resistance: We measure the collision rate of each NCHF, calculated as the ratio of the number of generated collisions to the total number of hashed keys.

- Avalanche effect: We use avalanche matrices (see Mulvey [2007] or Appleby [2008]) in which the probability of a change in each input-output pair of bits deviates from the ideal probability (0.5). We also use error measures (in terms of RMSE) of the complete avalanche matrices with respect to the ideal avalanche matrix.

- Speed: In this work the speed is considered as a requisite of NCHF, instead of a feature, so we do not perform speed comparisons. As we already stated in Section 5.1, individuals evolved by GP-hash have a limitation on the number of operations they can perform on each mixing cycle. This way, we only allow GP-hash to evolve efficient NCHF whose executions times are below a given threshold. The idea idea is to focus on evolving NCHF with good distribution, collisions, and avalanche properties, which are the real important properties, while keeping the execution time bounded.

6.1.2 Key sets

Jenkins Jenkins [1997] identified four patterns that usually appear in key sets. These patterns can be summarized as follows :

- Keys consist of common substrings arranged in different orders.
- Keys often differ with respect to only a few bits.
- The only difference between keys is that their lengths are different, i.e., "aaaa" vs. "aa".
- Keys are nearly all zeroes, and only a few bits are set to 1.

According to Jenkins' experience, most key sets, both human-selected and computer-generated, match at least one of these patterns.

Another interesting report on how to construct key sets for NCHF evaluation is Fai [1996], where the author T.C. Fai divides key sets into two classes: real sets, like those used in McKenzie et al. [1990], and synthetic sets. Inspired by a 1953 memorandum written by H. P. Luhn for IBM (which is considered by Donald E. Knuth to be the first hashing publication ever), Fai points out that the purpose of an NCHF is to disrupt any order or pattern that the keys could contain to generate the most random possible output. Thus, Fai deduces that the most difficult key sets should be those that are more compressible; i.e., those that contain the minimum amount of information, or the maximum amount of order.

Inspired by the ideas of Jenkins and Fai, we designed eight different key sets for our experiments: four *real*, and four generated synthetically for this work:

1. Real key sets:
    - **NAMES**: This set is a list issued by the government of the city of Buenos Aires, Argentina, which contains all of the names allowed for newborn babies. In addition to the HTML labels, each line contains a name, gender, the number of the act that regulates the name, and some optional information about its origin and meaning. Most of the characters in each line are HTML labels, which are almost the same in every line, and thus this set contains keys that are very similar (i.e., it contains very little information).
    - **PASSWD**: This is a huge text file (41Mb) that contains 3,721,256 common passwords, including alphanumeric combinations and words in 13 different languages. It is useful for testing the performance of NCHF against short alphanumeric strings, which are very common in hashing applications.
    - **MEGATAB**: This key set was extracted from an 18Gb MySQL table with 100,000,000 rows, each of which contains 26 different data points for a person: complete name, id number, gender, age, etc. To construct our key set, we randomly extracted 2000 rows from the table, and only used the following columns: first name, middle name, last name, nationality, gender and age. Key sets of this type that are comprised of aggregations of personal data are quite common in hashing applications.

- **LCC**: This set contains all of the compilation symbols that were created while compiling the source code of lcc, a retargetable compiler for ANSI C [Fraser et al., 1995]. Symbol tables for compilers and lexical analyzers are a paradigmatic application of NCHF.

2. Synthetic key sets:

   - **SPARSE**: This set contains 1000 bit strings of 128 bits each. The main feature of these keys is that they are almost all zeroes, and only a few bits are set to 1. They are created from a statistical distribution that sets the probability of a bit containing a 1 to less than an upper limit $\lambda = 0.1$.
   - **RANDOM**: This set contains 1000 strings of 128 bits. Each bit has a fixed probability of being set. The probability distribution is generated randomly in a previous stage, and used to produce all of the keys. This means that most of the bits are biased toward 1 or 0.
   - **REPEAT**: This set contains 1000 strings of 512 bits each. Keys of this set are strings composed of a set of substrings arranged in different orders. To create them, we selected 16 common English words and created a master string with them. All of the keys of this set are different permutations of this master string .
   - **LENGTH**: Keys of this set contain only 'a' characters and blank spaces in a 90:10 proportion, respectively . The set consists of 1000 keys of between 80 and 512 bits. Keys of this set only differ in length and the position of the spaces, which is consistent with the third pattern described by Jenkins. This presents a very difficult test for NCHF, which are expected to generate different hash values for very similar strings, like "aaaa" and "aa", or "aaaa aa" and "aa aaaa".

6.2 Results

First we show the results of GP-hash with the basic configuration. Then, we show how the sample size of the Monte Carlo simulation used in the avalanche fitness calculation can be raised to solve particular problems with special key sets.

6.2.1 Basic configuration

In the first place we examine the results of the stage 1, including the 50 GP-hash independent runs and the selection of the 5 best runs. Then, in stage 2, we compare the selected hashes with the state of the art in non cryptographic hashing.

Stage 1: Table 4 and Figure 6 summarize the results of the 50 GP-hash runs with the basic configuration obtained in the experiments of previous sections. All the 50 executions achieved very good fitness values, between 0.0448 and 0.0523. Differences between executions are very small. This suggest GP-hash is a very robust system: once the parameters have been correctly set, it is able to find NCHF with very good avalanche properties almost on every execution. We already observed this feature during the parameter tuning in Section 5.3. Figure

7 shows the evolution curves of the 50 GP-hash runs. Gray circles represent the fitness of the best individual of each run on each generation. The black curve represents the average of those fitnesses on each generation. As we already observed during the parameter tuning, GP-hash usually perform most of the fitness improvements during the earlier generations. In fact, we observe that around generation 20 all the runs already founded the basic structure of their best individuals. From generation 20 to the end of the run the computational effort is only devoted to fine grain adjustments.

	Min	**Average**	**Var (std. dev.)**	**Max**
Fitness	0.0448	0.0459	9.2×10^{-7} (0.0009)	0.0523
Nodes	16	21	7.6734 (2.7701)	25
Depth	9	12	4.8412 (2.2002)	17

Table 4: Summary of minimum, maximum, average, variance, and standard deviation values of fitness, number of nodes, and depth.

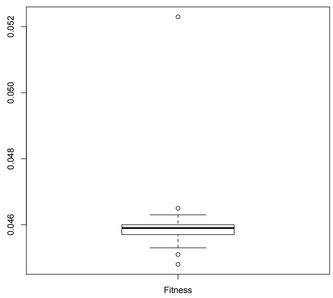


Figure 6: Boxplot of the fitness values of the 50 independent runs of GP-hash.

We select the best five GP-hash runs, and extract their best individuals. By wrapping these individuals into Merkle–Damgård constructions we obtain five fully functional NCHF. We label these hashes as gp-hash601, gp-hash602, gp-hash603, gp-hash604, and gp-hash605. We call this five NCHF the gp-hash600 family. The simplified pseudocode of their mixing functions is the following:

```
//gp-hash601:
        (Integer.rotateRight((hval * A0), 10) ^ (A0 * (hval ^ A0)));
//gp-hash602:
        (Integer.rotateRight(((hval ^ hval) ^ (A0 * hval)), 10)
        ^ ((hval ^ A0) * A0));
//gp-hash603:
        (((A0 ^ hval) * A0) ^ Integer.rotateRight((hval * A0), 8));
//gp-hash604:
        (Integer.rotateRight((A0 ^ (A0 * hval)), 7) ^ (A0 ^ (A0 * hval)));
//gp-hash605:
        (Integer.rotateRight((Integer.rotateRight((hval * A0), 4) ^ hval), 3)
        ^ ((hval ^ A0) * A0));
```

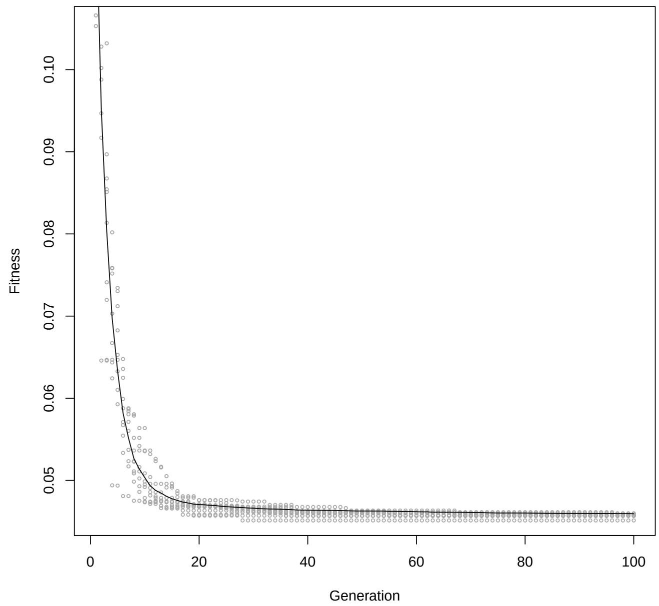


Figure 7: Evolution curve of the experiments with the basic configuration. Gray circles represent the fitness of the best individual of each GP-hash run on each generation, and black curve is the average of those fitness values.

Stage 2: Next step is to compare gp-hash600 hashes with the ten NCHF selected as benchmarking. We performed the avalanche, distribution, and collisions tests specified in Section 6.1, and results are very clear: hashes belonging to gp-hash600 family have outstanding avalanche properties, only comparable to the best NCHF of the state of the art (Figure 8 shows the avalanche error of each NCHF), but they are also highly competitive in terms of distribution of outputs and collisions resistance with all the specified key sets, with only one single exception: the SPARSE key set. Figure 9 show the Bhattacharyya distances of the distributions of outputs generated by each NCHF with each key set. With the SPARSE set most of the gp-hash600 hashes have serious problems, and generate very poorly distributed outputs. Results of the collisions tests (not shown here due to space limitations) are consistent with this observation: except for gp-hash605, all the other hashes of gp-hash600 family produce four times more collisions than the reference NCHF and, which is worst, they generate their most probable hash value for up to 487 different keys, when the average of the reference NCHF is only $4,3$ (more than 100 times better).

In the remainder of this section we focus on the problems with the SPARSE key set instead of analyzing in detail the complete results of gp-hash600 family. We are more interested in understanding and solving this problem before.

Problems with the SPARSE key set: Figure 10 shows the frequency of the hash values generated by lookup3 and gp-hash601. The differences are ob-

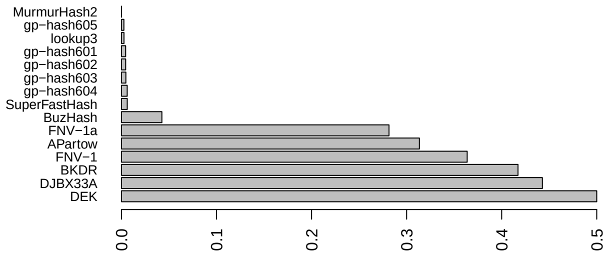


Figure 8: RMSE error between the avalanche matrices of each NCHF and the ideal avalanche matrix (that contains only 0.5 values). Hashes of the gp-hash600 family obtain avalanche scores only comparable to those obtained by SuperFastHash, lookup3, and MurmurHash2.

vious: while lookup3 generates very few times most of the possible hash values (four times maximum), gp-hash601 follows exactly the opposite approach: it generates most of the times a few oversampled hash values. More precisely, gp-hash601 generates 487 times the hash value `0x0`, and 70% of the input keys hashed to only five different hash values. It is significative that the most probable hash value is `0x0` when we are hashing a key set like SPARSE in which all the keys contains almost only zeroes (we already explained the construction of the key sets in Section 6.1.2).

The reason of this behavior is that the mixing function of most gp-hash600 hashes relies in a multiplication by the input byte of each step. If the last byte of a key is `0x0`, then the last step of the hash function multiplies the internal state by zero, and the result is always `0x0`. When dealing with a key set in which there are mostly only zeroes, this happens very often.

By design, the avalanche fitness function has the power to detect this kind of behaviors, penalizing them with poor scores. The problem is that the fitness function is not calculating the real avalanche error of the individuals, which will require to sample all the possible 32-bit input values. Instead, it estimates this error using a Monte Carlo simulation with a sample size that we initially set to $N = 100$ in Section 5.1.1. Considering that here are 2^{32} possible input bit strings, and only 2^{24} of them have only zeros on the last byte, when we randomly select 100 bit strings, in average we are only sampling 0.37 of zero-ended strings. This means that most of the times not even one of those bit strings has influence on the fitness calculation, and this avoids GP-hash to detect and penalize sensible hashes.

The solution we propose here is to increase the sample size to $N = 1000$, so the average number of zero-ended keys sampled raises to 3.7, which should be more than enough to detect and eliminate the problematic hashes. We analyze the results obtained with this configuration in next section.

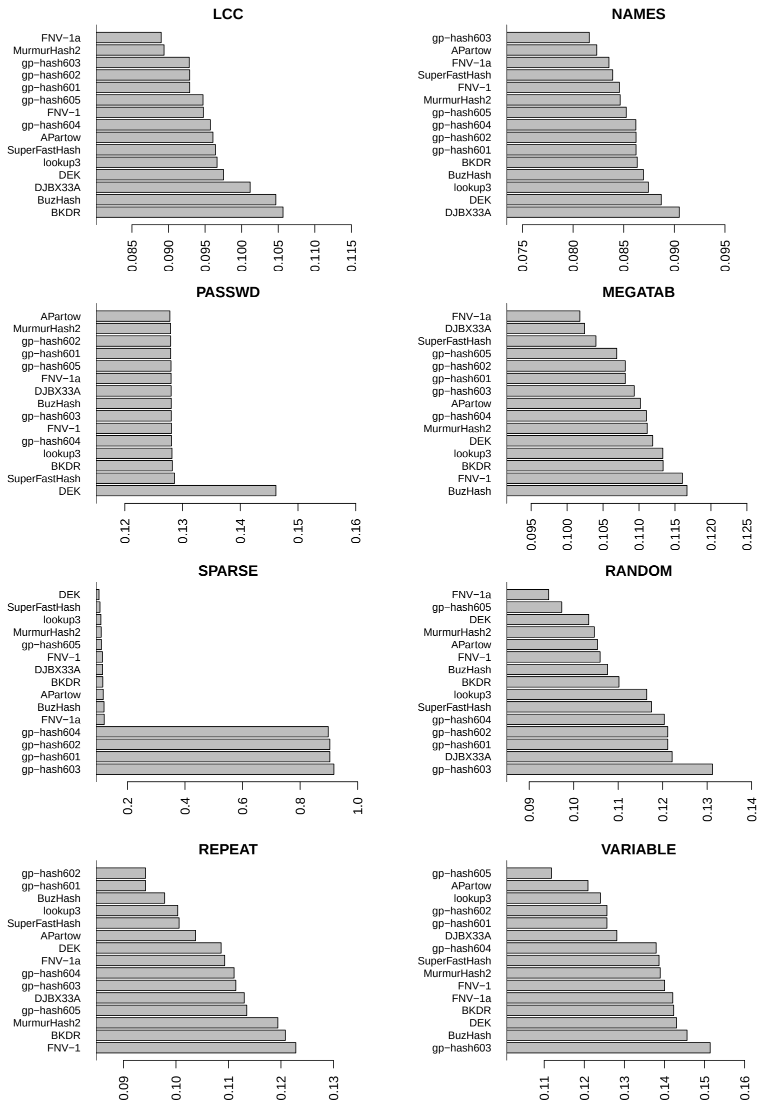


Figure 9: Bhattacharyya distance between the output distribution generated by each NCHF for each key set and the ideal uniform distribution (lower distances are better).

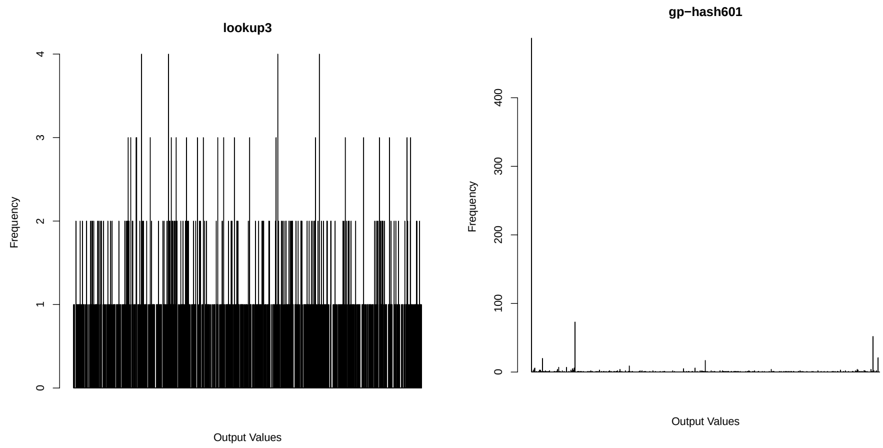


Figure 10: Frequency of hash values produced by lookup3 (left) and gp-hash601 (right).

6.2.2 Raising the sample size to $N = 1000$

In this experiments we use the same basic configuration than in previous section, but with two important differences: First, the sample size of the Monte Carlo simulation used to estimate avalanche fitness is raised to $N = 1000$. Second, in order to shorten the execution time of each experiment (the new sample size means that fitness calculations are 10 times slower) we reduced the number of generations to 50. This helps maintaining the efficiency of GP-hash while preserving most of its exploitation capabilities (the most important evolution always happens before generation 50, as we show in figures 7 and 12).

Stage 1: Table 5, and figures 11 and 12 summarize the results of the 50 GP-hash runs with the extended sample size. In this case, the evolution curve is even more abrupt, with most of the fitness improvements happening before generation 15. Furthermore, the variance of the fitness values is lower than with $N = 100$ (1.21×10^{-8} vs. $9{,}2\times10^{-7}$). This is a logical consequence of the greater sample size. Finally, we notice that the fitness values obtained with this new fitness configuration are considerably better than with the smaller sample size: average avalanche fitness is 0.0146 in this case, way better than the previous value of 0.0459.

	Min	**Average**	**Var (std. dev.)**	**Max**
Fitness	0.01431	0.0146	1.21×10^{-8} (0.0001)	0.01475
Nodes	18	22.56	3.84 (1.9596)	25
Depth	9	12.64	3.3233 (1.8230)	16

Table 5: Summary of minimum, maximum, average, variance, and standard deviation values of fitness, number of nodes, and depth.

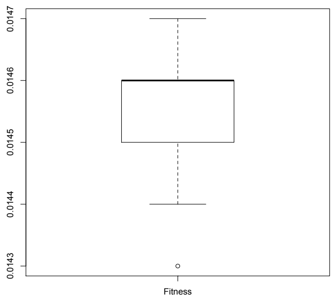


Figure 11: Boxplot of the fitness values of the 50 independent runs of GP-hash with $N = 1000$ and $G = 50$.

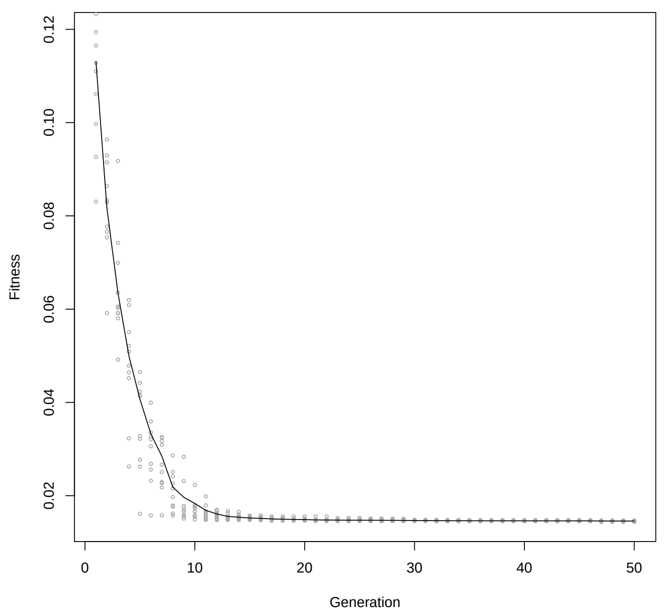


Figure 12: Evolution curve of the experiments with $N = 1000$ and $G = 50$. Gray circles represent the fitness of the best individual of each GP-hash run on each generation, and black curve is the average of those fitness values.

We select the best five GP-hash runs, and extract their best individuals. By wrapping these individuals into Merkle–Damgård constructions we obtain five fully functional NCHF. We label these hashes as gp-hash611, gp-hash612, gp-hash613, gp-hash614, and gp-hash615. We call this five NCHF the gp-hash610 family. The simplified pseudocode of their mixing functions is the following:

```
//gp-hash611:
  (Integer.rotateRight((hval ^ Integer.rotateRight(((hval ^ (hval ^ A0))
  * Integer.rotateRight(hval, 1)), 4)), 3) ^ (A0 * (hval ^ A0)));
//gp-hash612:
  (((A0 ^ hval) * hval) ^ (Integer.rotateRight((hval * A0), 13) ^ A0));
//gp-hash613:
  (Integer.rotateRight(((A0 * hval) ^ Integer.rotateRight(hval, 1)), 7)
  ^ (A0 * (A0 ^ hval)));
//gp-hash614:
  (((hval ^ A0) * (A0 ^ hval)) ^ Integer.rotateRight((A0 * hval), 7));
//gp-hash615:
  ((A0 * (hval ^ A0)) ^ Integer.rotateRight((hval * (hval ^ A0)), 10));
```

Stage 2: The gp-hash610 family obtains even better results in the avalanche tests than gp-hash600. The avalanche matrices of the gp-hash610 hashes (Figure 13) are almost perfect, with all the squares close to pure white. In fact, the total error (in terms of RMSE) of their avalanche matrices are between 0.0022 and 0.0011 (see Figure 14). Only MurmurHash2, the most powerful NCHF in terms of avalanche effect, is able to outperform gp-hash610 functions in the avalanche tests.

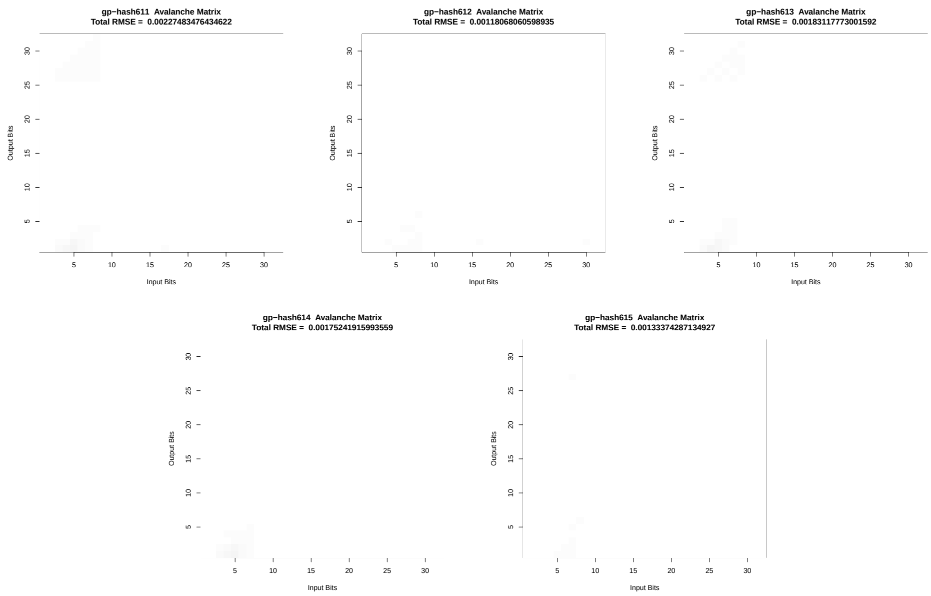


Figure 13: Avalanche matrices of functions gp-hash611, gp-hash612, gp-hash613, gp-hash614 y gp-hash615.

Furthermore, the results of the Bhattacharyya distance tests (Figure 15) show that gp-hash610 functions are also competitive in terms of distribution of

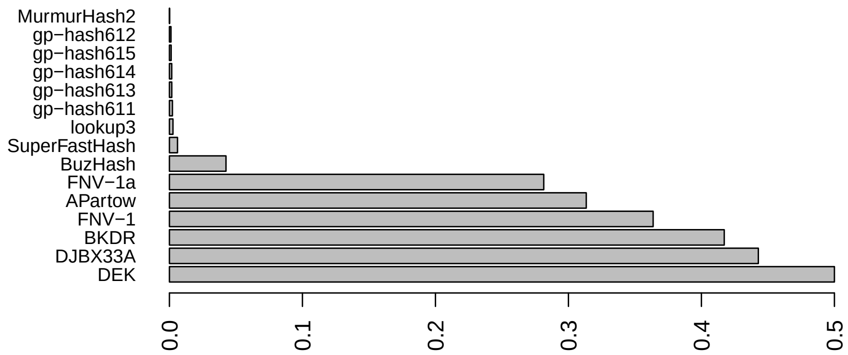


Figure 14: RMSE error between the avalanche matrices of each NCHF and the ideal avalanche matrix (that contains only 0.5 values). Hashes of the gp-hash610 family obtain avalanche scores only comparable to those of MurmurHash2, the NCHF with the best avalanche properties.

outputs: gp-hash612 is the best function to hash the key sets LCC, NAMES, and RANDOM, while gp-hash611 is the best NCHF available for the key set SPARSE. Other functions like gp-hash615 or gp-hash614 also deliver very decent distributions on some key sets, in which they are the second best NCHF. On the other hand, some gp-hash610 functions perform poorly in some sets. This is the case of gp-hash615 on MEGATAB and VARIABLE sets, or gp-hash613 on LCC.

It is also important to note that the larger sample size used to evolve gp-hash610 family obviously improves the performance os GP-hash with the SPARSE key set. Although gp-hash614 still have problems with zero-ended strings, gp-hash613 obtains competitive results, and there is even one hash, gp-hash611, that achieves in this set the best distribution among all the NCHF tested.

Results of the collisions tests (shown in Figure 16) are even better: on seven of the eight key sets tested, a function belonging to the gp-hash610 family is the best in terms of collisions rate. The only key set in which gp-hash610 does not win is RANDOM, but in this case, gp-hash612, gp-hash614, and gp-hash613 are the second, third, and fourth best function respectively.

6.2.3 Discussion

Functions of the gp-hash600 family obtain interesting results. They show very high levels of avalanche effect, only comparable to those obtained by lookup3, SuperFastHash, and MurmurHash2. They also obtain competitive results in the distribution and collisions tests. However, they have a flaw that greatly

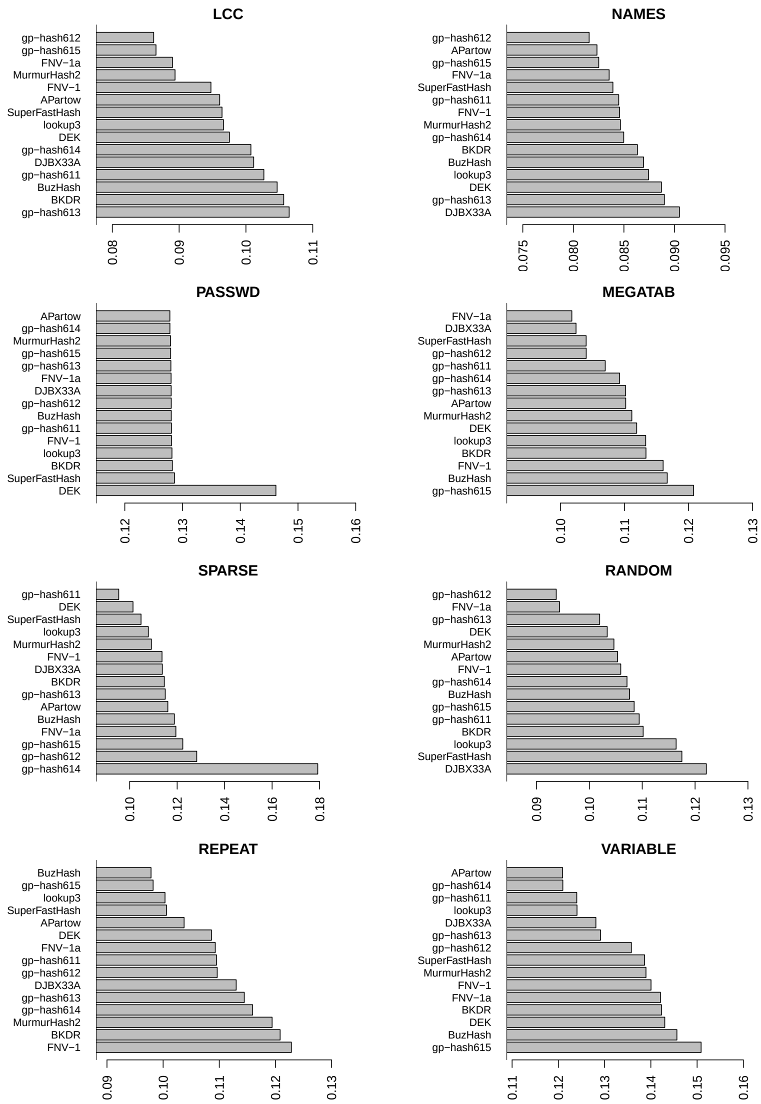


Figure 15: Bhattacharyya distance between the output distribution generated by each NCHF for each key set and the ideal uniform distribution (lower distances are better).

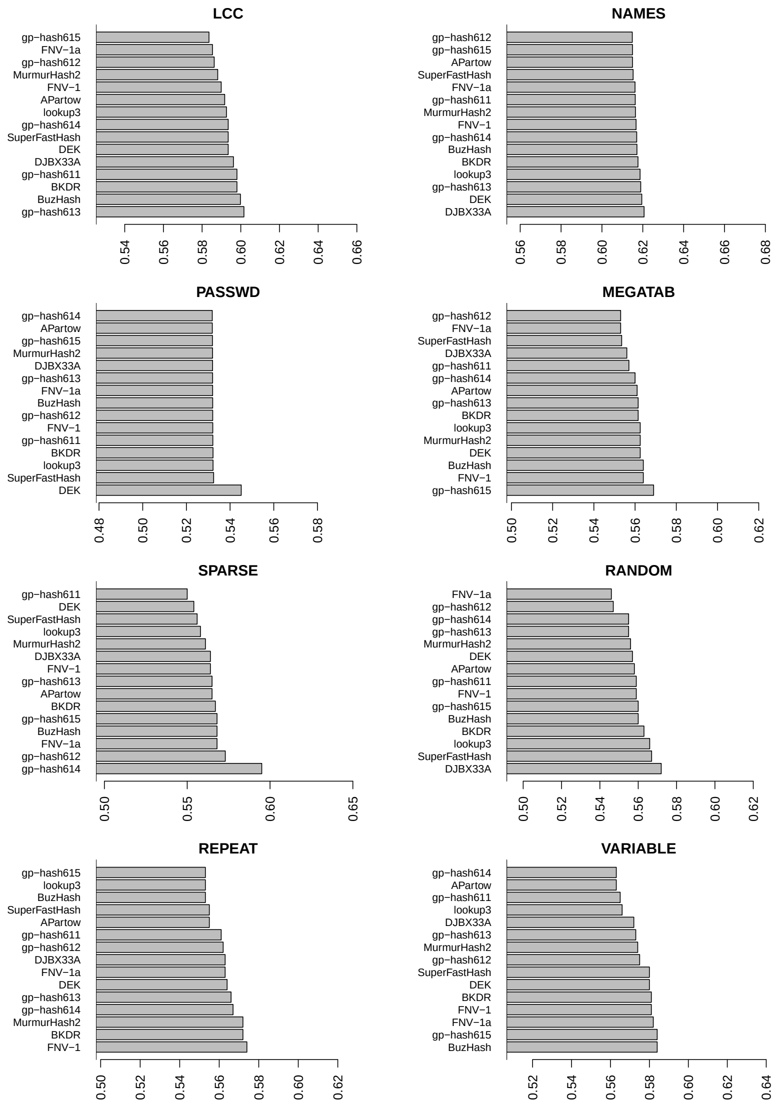


Figure 16: Collisions rate of each NCHF for each key set (lower collisions rates are better).

affects their performance with the SPARSE key set, which contains a very high proportion of zero-ended keys. This flaw is a consequence of the reduced sample size ($N = 100$) used in the Monte Carlo simulation that estimates the avalanche fitness. With $N = 100$, it is very unlikely that any zero-ended string is sampled during the fitness evaluation, so the GP-hash system is unable to exert an evolutionary pressure towards unflawed individuals.

Increasing this sample size to $N = 1000$ has proven to be effective: the gp-hash610 family is not generally affected by this problem, and in fact, gp-hash611 has proven to be a very good choice for hashing the SPARSE key set. From a practical point of view, the sample size should be adjusted according to the probability of the patterns contained in the key sets of our particular application.

Finally, we want to stress that, in general, gp-hash610 functions are competitive with the state of the art in non cryptographic hashing, delivering outstanding avalanche properties, competitive results in the distribution of outputs tests, and exceptional collisions resistances. We want to stress that this functions were automatically designed by GP-hash, and we can safely claim that they are *at least as good* as a selection of the best NCHF created by human experts.

7 Conclusions

Hashing is of capital importance in the software industry. The possibility of finding objects in a set in constant $O(1)$ time, independently of the size of the set, is essential for software engineers. However, very often they do not pay enough attention to the critical process of designing appropriate hash functions for their particular problems. This is understandable: designing good hash functions is a difficult process due to the extremely nonlinear constructions they use. Hash functions are designed in such a way that humans can not easily invert them, so it is perfectly natural that these expressions are difficult to design.

However, the same design principles that makes this process difficult for humans, also seems to make it very suitable for GP: Highly non-linear domains, in which the interrelationships among the relevant variables is unknown or not completely understood, are precisely the most adequate for GP, as stated in [Poli et al., 2008].

Surprisingly, there is not much research about the application of GP, Evolutionary Computation, or Artificial Intelligence to the design of good NCHF. In section 4, we reviewed the most interesting papers on this topic that we know of. The approaches of those works have some merit, but we still think that this topic really worths a lot more research.

The central claim of this work is that it is possible to use GP to substitute human experts in the challenging task of designing high-quality, general purpose NCHF. For this task, we created the GP-hash system, and we learned some important facts in the process.

The most important difference with other works on the application of Evolutionary Computation to the automatic design of hashes is the fitness function. Previous approaches invariably use fitness functions based on the collisions resistance of the evolved hashes, which is a problem, because collisions properties are data dependent: we can only measure the collisions resistance of a hash with relation to a specific key set. This lack of generality is a major drawback when evolving general purpose NCHF, which are expected to deliver a proper performance with a great number of very different key sets. On the other hand, avalanche effect is a statistical measure of an intrinsic property of the mixing function, and thus, it is completely independent of the hashed key set. Furthermore, this property is also a measure of the ability of the hash to disrupt the input patterns, and to efficiently spread the input bits over the internal state, producing an apparently unpredictable output. These concepts are closely related to a good distribution of outputs (the more random the output looks, the more evenly the outputs distribute) and thus with the collisions rate (biased distributions generate more collisions than pure uniform distributions). Based on this, we hypothesize that avalanche effect could be a very good estimator of the overall quality of a NCHF. And the results shown in Section 6.2 seem to support this claim: hashes evolved with avalanche fitness have outstanding avalanche properties, and they also perform very well in the distribution and collisions tests.

Concerning the terminals and functions set, we gathered together ten of the most important functions of the hashing literature and of the software industry. We studied the operators and variables they use to generate a basic terminal and function set, and then we applied a methodology similar to [Wang and Soule, 2004] to refine this set. We discovered some interesting facts: First, that magic constants are not needed to evolve hashes with high avalanche effect; Second, that two very popular operators like addition and *xor* form a group, and only one of them is needed (this is intuitively supported by the fact that hash functions that do not use addition, always use *xor*, and vice versa). These two discovers could help other researchers that want to apply Evolutionary Algorithms to hashing, but they also suggest hashing experts that magic constants may not be necessary in the construction of non cryptographic hashes.

We also found out that GP-hash system is highly robust, and can work well with very different parameter configurations. This also supports the accepted idea that GP is a very robust technique in general.

Finally, we experimentally demonstrate the utility of GP-hash by generating a set of new general purpose NCHF that are competitive with the state of the art in non cryptographic hashing. We used GP-hash to generate two different families of NCHF. We call them gp-hash600 and gp-hash610. The functions of the former have a weakness that makes them fail when hashing zero-ended keys, but this flaw was addressed and solved in the gp-hash610 family. Functions belonging to this family have outstanding avalanche properties, only surpassed by MurmurHash2, the NCHF with the best avalanche properties of the state of the art. Furthermore, gp-hash610 hashes are competitive as well in terms of collisions resistance and distribution of outputs with a selection of the ten most widely-used NCHF of the literature. All these facts supports the central claim of this work: that GP, when using the avalanche fitness and an appropriate functions and terminals set, is able to generate non cryptographic hash functions that are similar to those generated by hashing experts with years of experience.

8 Acknowledgments

This work have been funded by spanish Department of Science and Innovation (Ministerio de Ciencia e Innovación), under the research project *Gestión de Movilidad Eficiente y Sostenible* (TIN2011-28336).

References

Aherne, F. J., Thacker, N. A. and Rockett, P.: 1998, The bhattacharyya metric as an absolute similarity measure for frequency coded data, *Kybernetika* **34**(4), 363–368.

Appleby, A.: 2008, Murmurhash 2.0.
URL: *http://code.google.com/p/smhasher/wiki/MurmurHash2*

Bellare, M. and Kohno, T.: 2004, Hash function balance and its impact on birthday attacks, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, Proceedings*, Vol. 3027 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 401–418.

Berarducci, P., Jordan, D., Martin, D. and Seitzer, J.: 2004, GEVOSH: Using grammatical evolution to generate hashing functions, *2004 Genetic and Evolutionary Computation Conference (GECCO 2004) Workshop Proceedings*, Seattle, Washington, USA.

Bertoni, G., Daemen, J., Peeters, M. and Assche, G. V.: 2008, On the indifferentiability of the sponge construction, *in* N. P. Smart (ed.), *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, Vol. 4965 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 181–197.

Bertoni, G., Daemen, J., Peeters, M. and Van Assche, G.: 2007, Sponge functions, *ECRYPT (European Network of Excellence for Cryptology) Hash Workshop, Barcelona, Spain, May 24-25.*

Biham, E. and Dunkelman, O.: 2006, A framework for iterative hash functions-haifa, *Second NIST Cryptographic Hash Workshop*, Vol. 2006, p. 2.

Caldwell, C.: 1994-2009, The prime pages.
URL: *http://primes.utm.edu/*

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.: 2001, *Introduction to algorithms*, The MIT Press, Cambridge, MA.

Crosby, S. A. and Wallach, D. S.: 2003, Denial of service via algorithmic complexity attacks, *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, USENIX Association, Berkeley, CA, USA, pp. 3–3.

Damgård, I.: 1990, A design principle for hash functions, *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology, London, UK*, Springer Berlin/Heidelberg, pp. 416–427.

Damiani, E., Liberali, V. and Tettamanzi, A.: 1998, Evolutionary design of hashing function circuits using an fpga, *ICES '98: Proceedings of the Second International Conference on Evolvable Systems, Lausanne, Switzerland, September 23-25*, Springer-Verlag, pp. 36–46.

Damiani, E. and Tettamanzi, A. G. B.: 1999, On-line evolution of fpga-based circuits: A case study on hash functions, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, Pasadena, California, July 19-21*, IEEE Computer Society, pp. 26–33.

Estébanez, C., Castro, J. C. H., Ribagorda, A. and Isasi, P.: 2006a, Evolving hash functions by means of genetic programming, *in* M. Cattolico (ed.), *Proceedings of the 8th Genetic and Evolutionary Computation Conference, GECCO 2006, Seattle, Washington, USA, July 8-12*, ACM Press, New York, pp. 1861–1862.

Estébanez, C., Castro, J. C. H., Ribagorda, A. and Isasi, P.: 2006b, Finding state-of-the-art non-cryptographic hashes with genetic programming, *in* T. P. Runarsson, H.-G. Beyer, E. K. Burke, J. J. M. Guervós, L. D. Whitley and X. Yao (eds), *Parallel Problem Solving from Nature - PPSN IX, 9th International Conference, Reykjavik, Iceland, September 9-13*, Vol. 4193 of *Lecture Notes in Computer Science*, Springer, pp. 818–827.

Estévez-Tapiador, J. M., Castro, J. C. H., Peris-Lopez, P. and Ribagorda, A.: 2008, Automated design of cryptographic hash schemes by evolving highly-nonlinear functions, *Journal of Information Science and Engineering* **24**(5), 1485–1504.

Fai, M. T. C.: 1996, *General Hashing*, PhD thesis, University of Auckland.

Feistel, H.: 1973, Cryptography and computer privacy, *Scientific American* **228**(5), 15–23.

Fowler, G., Vo, P. and Noll, L. C.: 1991, Fowler / noll / vo (fnv) hash.
URL: *http://isthe.com/chongo/tech/comp/fnv/*

Fraser, C., Hansen, D. and Hanson, D.: 1995, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA.

Goodrich, M. T. and Tamassia, R.: 2009, *Algorithm Design: Foundations, Analysis and Internet Examples*, John Wiley & Sons, Inc., New York, NY, USA.

Gordon, T. G. W. and Bentley, P. J.: 2002, On evolvable hardware, *Studies in Fuzziness and Soft Computing* **101**, 279–323.

Heileman, G. L.: 1996, *Data structures, algorithms and object-oriented programming*, McGraw-Hill, New York, NY.

Henke, C., Schmoll, C. and Zseby, T.: 2008, Empirical evaluation of hash functions for multipoint measurements, *SIGCOMM Computer Communication Review* **38**(3), 39–50.

Hsieh, P.: 2004-2008, Hash functions.
URL: *http://www.azillionmonkeys.com/qed/hash.html*

Hussain, D. and Malliaris, S.: 2000, Evolutionary techniques applied to hashing: An efficient data retrieval method, *in* L. D. Whitley, D. E. Goldberg, E. Cantú-Paz, L. Spector, I. C. Parmee and H.-G. Beyer (eds), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00), Las Vegas, Nevada, USA, July 8-12, 2000*, Morgan Kaufmann, p. 760.

Jenkins, R. J.: 1997, Hash functions for hash table lookup, *Dr. Dobb's Journal* .
URL: *http://burtleburtle.net/bob/hash/evahash.html*

Kernighan, B. W. and Ritchie, D.: 1988, *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ.

Klink, A. and Wälde, J.: 2011, Effective denial of service attacks against web application platforms, Talk at 28th Chaos Communication Congress (28C3).

Knott, G. D.: 1975, Hashing functions, *The Computer Journal* **18**(3), 265–278.

Knuth, D. E.: 1973, *The Art of Computer Programming, Volume III: Sorting and Searching*, Addison-Wesley, Reading, MA.

Knuth, D. E.: 1998, *The Art of Computer Programming, Volume III: Sorting and Searching, Second Edition*, Addison-Wesley Professional.

Koza, J. R.: 1992, *Genetic Programming: On the Programming of Computers by Natural Selection*, MIT Press, Cambridge, MA.

Lucks, S.: 2005, A failure-friendly design principle for hash functions, *in* B. K. Roy (ed.), *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, Vol. 3788 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 474–494.

Luke, S.: 2001, When short runs beat long runs, *in* L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon and E. Burke (eds), *Proceedings of the Third Genetic and Evolutionary Computation Conference (GECCO 2001), San Francisco CA*, Morgan Kaufmann Publishers, Inc., pp. 74–80.

Matsui, M. and Fukuda, S.: 2005, How to maximize software performance of symmetric primitives on pentium III and 4 processors, *in* H. Gilbert and H. Handschuh (eds), *Fast Software Encryption, 12th International Workshop, FSE 2005, Paris, France, February 21-23*, Vol. 3557 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 398–412.

McKenzie, B. J., Harries, R. and Bell, T.: 1990, Selecting a hashing algorithm, *Software: Practice and Experience* **20**(2), 209–224.

Merkle, R. C.: 1989, One way hash functions and des, *CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, Proceedings*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 428–446.

Miyaguchi, S., Ohta, K. and Iwata, M.: 1990, 128-bit hash function (n-hash), *NTT Review* **2**(6), 128–132.

Mulvey, B.: 2007, Hash functions.
URL: *http://home.comcast.net/ bretm/hash/*

O'Neill, M. and Ryan, C.: 2003, *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, Vol. 4 of *Genetic programming*, Kluwer Academic Publishers.

Partow, A.: 2010, General purpose hash function algorithms.
URL: *http://www.partow.net/programming/hashfunctions/*

Poli, R., Langdon, W. and McPhee, N.: 2008, *A field guide to genetic programming*, Lulu Enterprises UK Ltd.

Preneel, B.: 1993, *Analysis and design of cryptographic hash functions*, PhD thesis, Katholieke Universiteit Leuven.

Ramakrishna, M. V. and Zobel, J.: 1997, Performance in practice of string hashing functions, *in* R. W. Topor and K. Tanaka (eds), *Database Systems for Advanced Applications '97, Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, April 1-4*, Vol. 6 of *Advanced Database Research and Development Series*, World Scientific, pp. 215–224.

Ryan, C., Collins, J., Collins, J. and O'Neill, M.: 1998, Grammatical evolution: Evolving programs for an arbitrary language, *Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming*, Springer-Verlag, pp. 83–95.

Safdari, M.: 2009, Evolving universal hash functions using genetic algorithms, *in* F. Rothlauf (ed.), *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, ACM, New York, NY, USA, pp. 2729–2732.

Schneier, B.: 1996, *Applied Cryptography, Second Edition*, John Wiley & Sons.

Sedgewick, R.: 2001, *Algorithms in C++, Third edition*, Addison Wesley, New Delhi.

Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Perez-Uribe, A. and Stauffer, A.: 1997, A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems, *IEEE Transactions on Evolutionary Computation* **1**(1), 83 –97.

Snasel, V., Abraham, A., Dvorsky, J., Ochodkova, E., Platos, J. and Kromer, P.: 2009, Searching for quasigroups for hash functions with genetic algorithms, *World Congress on Nature Biologically Inspired Computing, NaBIC 2009, Coimbatore, India, December 9-11*, pp. 367 –372.

Valloud, A.: 2008, *Hashing in Smalltalk: Theory and Practice*, self-published (www.lulu.com).

Wang, G. and Soule, T.: 2004, How to choose appropriate function sets for gentic programming, *in* M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa and T. Soule (eds), *Genetic Programming 7th European Conference, EuroGP 2004, Coimbra, Portugal, April 5-7, Proceedings*, Vol. 3003 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 198–207.

Wang, T.: 2007, Integer hash function.
URL: *http://www.concentric.net/ Ttwang/tech/inthash.htm*

Webster, A. F. and Tavares, S. E.: 1986, On the design of s-boxes, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, Proceedings*, Vol. 218 of *Lecture Notes in Computer Science*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 523–534.

Widiger, H., Salomon, R. and Timmermann, D.: 2006, Packet classification with evolvable hardware hash functions - an intrinsic approach, *in* A. J. Ijspeert, T. Masuzawa and S. Kusumoto (eds), *Biologically Inspired Approaches to Advanced Information Technology, Second International Workshop, BioADIT 2006, Osaka, Japan, January 26-27, 2006, Proceedings*, Vol. 3853 of *Lecture Notes in Computer Science*, Springer, pp. 64–79.