Estébanez, C., Saez, Y., Recio, G. and Isasi, P. (2014), Performance of the most common non-cryptographic hash functions. Softw. Pract. Exper., 44: 681-698

DOI: https://doi.org/10.1002/spe.2179

Performance of the most common non-cryptographic hash functions

César Estébanez*,†, Yago Saez, Gustavo Recio and Pedro Isasi

Department of Computer Science, Universidad Carlos III de Madrid, Av. Universidad 30, 28911, Leganes, Madrid, Spain

SUMMARY

Non-cryptographic hash functions (NCHFs) have an immense number of applications, ranging from compilers and databases to videogames and computer networks. Some of the most important NCHF have been used by major corporations in commercial products. This practical success demonstrates the ability of hashing systems to provide extremely efficient searches over unsorted sets. However, very little research has been devoted to the experimental evaluation of these functions. Therefore, we evaluated the most widely used NCHF using four criteria as follows: collision resistance, distribution of outputs, avalanche effect, and speed. We identified their strengths and weaknesses and found significant flaws in some cases. We also discuss our conclusions regarding general hashing considerations such as selection of the compression map. Our results should assist practitioners and engineers in making more informed choices regarding which function to use for a particular problem. Copyright © 2013 John Wiley & Sons, Ltd.

KEY WORDS: Non-cryptographic hash functions; avalanche effect; avalanche matrices; collision resistance; distribution of outputs; modulo prime

1. INTRODUCTION

Non-cryptographic hash functions (NCHFs) play an important role in computer science. Indeed, hash tables, which are their most widely known application, are present in thousands of different software systems. NCHF also have many other practical applications, including lexical analyzers and compilers, databases, communication networks, bloom filters, videogames, DNS servers, filesystems, and virtually any piece of code that requires information to be looked up very quickly. Ideally, NCHF makes it possible to search for objects in a set within a constant time $O(1)$, regardless of the size of the set. In addition to providing optimal access times, NCHF, again ideally, confers perfect scalability to hashing systems.

In the era of the Internet and ubiquitous computing, corporations need to manage very large databases that are continually growing, often on an exponential basis. For example, as of April 2009, users of Facebook had uploadeded over 15 billion photos, which required 1.5 petabytes of storage, and this value was expected to grow by 25 terabytes each week [1]. The Large Hadron Collider (LHC) at Conseil Européen pour la Recherche Nucléaire (CERN) generates 10 terabytes of data every 8 hours of running time, and, at full power, the LHC could produce 10 petabytes of useful data each year [2]. The bidding portal eBay maintains a data warehouse that contains 6.5 petabytes of user data and 17 trillion records and processes 150 billion new records every day (around 50 terabytes consumed per day) [3]. Finally, Google processed 20 petabytes of data every day in 2008 [4].

*Correspondence to: César Estébanez, Office 2.2.A06, Sabatini Building, Universidad Carlos III de Madrid, Av. Universidad 30, 28911, Leganes, Madrid, Spain.

†E-mail: cesteban@inf.uc3m.es

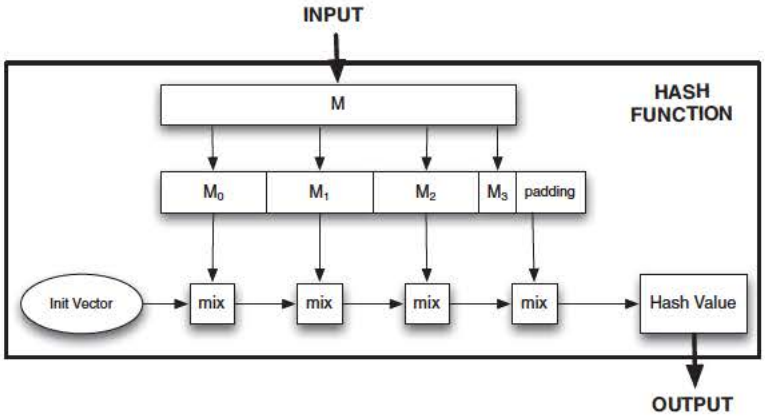

Figure 1. Merkle–Damgård construction.

In this context, NCHF, with their capability to perform very fast searches in sets regardless of their size, should play a very important role, as suggested by the impressive list of their real applications: Linux and FreeBSD distributions, Twitter, DNS servers, NFS implementations, video games, implementations of PostgreSQL, Linux, Perl, Ruby, or Infoseek, programming languages such as PHP 5, Python, and ASP.NET.

However, very little scientific work has been devoted to the empirical evaluation of NCHF. In this work, we experimentally studied the performance of the most widely known NCHF in the literature, identified their main features and their strengths and weaknesses, and point out important flaws in some cases. We considered the most accepted quality criteria for an NCHF: avalanche effect, distribution of outputs, collision resistance, and speed. The results of this study could help practitioners and engineers to select the most suitable function for their particular problems and should be of great interest to the global hashing community.

1.1. Non-cryptographic hash functions

Non-cryptographic hash functions are a family of mathematical functions that take as input a message of variable size and return a hash value of constant size (usually smaller than the message) that ideally identifies the message in a one-to-one correspondence. This condition is obviously impossible because the number of potential inputs is infinite and the number of potential outputs is finite. However, even when collisions (where two different keys generate the same hash value) are unavoidable, they should be minimized for NCHF to be useful.

Almost every NCHF is constructed following the same scheme, known as *Merkle–Damgård construction*[‡]. This scheme consists of dividing the input message into pieces of constant size and processing these pieces one by one using a mixing function, which combines each piece with its internal state to generate highly entropic outputs.

Figure 1 shows this process.

Hash functions usually consist of four fundamental blocks.

- Initialization block.
- Mixing block, in which the input message is split into pieces and processed step by step.
- Finalization block.
- Compression block, or compression map if we follow the nomenclature of [5].

[‡]This does not apply to cryptographic hash functions that use a variety of systems other than Merkle–Damgård. This is because this construction scheme is no longer considered safe, because different cryptanalysis studies have exposed some weaknesses that are considered important for cryptographic applications. Alternative schema include HAsh Iterative FrAmework (HAIFA) [6], *wide-pipe construction* [7], and *sponge construction* [8,9].

DOI: 10.1002/spe

The initialization and finalizatio blocks are optional. As noted earlier, in the mixing block, the input message is split into pieces that are processed one by one. The compression block can be considered to be part of the hash function itself, as in [5], or a completely independent subsystem, which is what most experts think. This block is necessary because, for an NCHF that generates outputs of size n bits, the hash values generated would be within the interval $[0, 2^n)$. However, in real applications, we usually need values within a smaller interval $[0, M)$, where $M < 2^n$. The compression map is the subsystem that is in charge of translating values $h \in [0, 2^n)$ into $h' \in [0, M)$.

The most straightforward method for performing this compression is to take the hash value produced, h, and use the modulo operation to scale it. In this manner, we force h' to be bounded by M by means of the operation $h' = h \ mod \ M$. The use of this compression method is almost unanimously accepted in the hashing literature and could be considered a standard. Nevertheless, controversy arises when trying to choose the best possible value for M. Some experts claim that M should be a prime number, whereas others argue that it is much better to use a power of two instead. Section 3.3.1 addresses this topic in greater detail.

1.2. 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** [10]. 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 efficien 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). There are two versions of this hash: FNV-1 and FNV-1a.
- **lookup3** [11]. This function was designed by Robert Jenkins and is one of the most important references in the fiel 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** [12]. 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 in turn is used in browsers such as Safari and Google Chrome. This function was also part of several versions of the former Macromedia product Flash Player.
- **MurmurHash2** [13]. 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, such as 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 influence many application servers, such as Tomcat, Geronimo, Jetty or Glassfish which could be exposed to Denial of Service attacks that use known weaknesses of DJBX33A to bring the application server to its knees (see [14] and [15]).
- **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 one and the other half have 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 [16]. 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 [17].

- **BKDR**. This function was originally proposed in [18] and is included in the aforementioned 'General Hash Function Library'.
- **APartow** [17]. This hybrid rotative and additive hash function algorithm was proposed by Arash Partow and is included in his library of hashes.

1.3. Purpose and organization

The main purpose of this work was to perform a comprehensive study of the properties of the functions that form the current state of the art in non-cryptographic hashing. We use the four most widely accepted quality criteria for NCHF and specifi metrics to discover strengths and weaknesses and report important limitations of some of the functions studied. We also recommend some NCHF and discourage the use of others for different specifi scenarios. This could help practitioners and researchers make more informed choices when implementing particular hashing systems.

The paper is organized as follows. Section 2 explains the criteria used to measure the quality of the functions studied. Section 3 presents the method used in the experimental part of the work. Section 4 shows the results of the experiments. Finally, Section 5 contains a discussion of the results and conclusions.

2. 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 [5, 19, 20]).

- **Collision resistance**. A hash function must reduce the collisions it produces to a minimum [5, 19, 21]. 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 fin two colliding keys using a *birthday attack*. However, it could take much fewer if the NCHF is poorly designed [22]. 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 specifi key set [16, 19].
- **Distribution of outputs**. It is very important for a non-cryptographic hash to produce outputs that follow a uniform distribution [19, 21, 23, 24]. 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 [19, 20, 25]. 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 simplifie its study and measurement.
- **Speed**. Non-cryptographic hash functions are useful because they allow searches to be performed very quickly. This means that a NCHF must be as fast as possible [5, 16, 23, 26–28]. For this purpose, NCHF should use very few operators, and these operators should be efficien in terms of CPU consumption. This criterion obviously depends on the architecture in which the hash function runs, because different CPUs offer different performance levels for the same operators (see, for example, [29]). The use of modern processors could speed up functions as well if their algorithms can be processed using Single Instruction, Multiple Data (SIMD) instructions.

3. METHODOLOGY

In this section, we describe the method used in our experiments. First, we introduce the metrics and tests used to measure each quality criterion. We then explain how we chose the key sets used in the

data-dependent tests. Finally, we will discuss the different output sizes and table configuration used in the experiments.

3.1. Metrics

We performed one test for each NCHF quality criterion.

- Distribution of outputs. We use the Bhattacharyya distance [30] as a measure of how close the outputs of a NCHF are to the ideal uniform distribution.
- 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 [31] or [13]) 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. We calculate how much information the function can hash per time unit.

3.2. Key sets

Jenkins [11] identifie 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, that is, 'aaaa' versus '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 [32], where the author T. C. Fai divides key sets into two classes: real sets, like those used in [28] and [33], 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 firs 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 difficul key sets should be those that are more compressible; that is, 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 fil (41 Mb) 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 18-Gb MySQL table with 100,000,000 rows, each of which contains 26 different data points for a person: complete name, id number, gender, age, and so on. To construct our key set, we randomly extracted 2000

[§]RMSE stands for Root Mean Square Error. It is define as the as the square root of the Mean Square Error (MSE).

rows from the table and only used the following columns: firs 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 [34]. 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 1 to less than an upper limit $\lambda = 0.1$.
    - **RANDOM**. This set contains 1000 strings of 128 bits. Each bit has a fi ed 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 difficul test for NCHF, which are expected to generate different hash values for very similar strings, such as 'aaaa' and 'aa', or 'aaaa aa', and 'aa aaaa'.

3.3. Table configu ations and output sizes

For the data-dependent metrics (collision resistance and distribution of outputs), we need to simulate the use of a hash table to obtain collisions and distribution statistics. Therefore, two parameters of hash tables must be define before these metrics can be used: load factor and table size.

The load factor is the ratio of the number of objects hosted by the table to the number of available buckets. The size of the table is often calculated by multiplying the desired load factor by the number of objects we expect to place in the table and then looking for the closest number that is either a prime number or a power of two.

To provide the most general results possible, we considered for our experiments four combinations of load factors and output sizes as follows: POWER-SPARSE, POWER-DENSE, PRIME-SPARSE, and PRIME-DENSE. SPARSE configuration use a load factor $\alpha \approx 0.5$, whereas DENSE configura tions use $\alpha \approx 2$. On the other hand, POWER configuration use the power of two that is closest to the product of multiplying the load factor by the size of the hashed key set, whereas PRIME configuration do the same with prime numbers instead of powers of two.

To identify the most interesting configuratio for our experiments, we performed two different tests.

1. Output size sensitivity test. We wanted to know if the use of prime numbers or powers of two produced a measurable difference in the performance of the NCHF studied.
2. Load factor sensitivity test. The purpose of this test was to determine if there is any difference in the performance of hash functions between sparse tables and dense tables.

3.3.1. Output size tests and the modulo prime myth. For the past few years, there has been a notable controversy in the hashing community regarding how to design the compression map of an NCHF. The polemic consists of whether the output size of the function (M) should be a prime number or a power of two. Supporters of the former assert that the use of prime numbers is of fundamental importance. They state that the goal is to eliminate possible biases inherited from the distribution

of the hashed keys. This idea seems to be inspired by the work of Knuth, who claimed that it was important to choose a prime number when using the modulo operation as a hashing function [16].

However, some engineers, professors, and practitioners who visit forums related to hashing have begun to question this idea (to the point of referring to it as *the modulo prime myth*), which has been repeated as a mantra for the past 30 years, although its practical usefulness has not been formally proven. The argument against the *modulo prime myth* is that the improvement in the output distribution that could be obtained by the use of prime numbers (if any) will be worthless because of the great inefficien y of the $mod\ M$ operation, where M is a prime number. In contrast, if M was a power of two, 2^x, we could eliminate the $mod\ 2^x$ operation and instead simply select the least significan x bits of the output, which is far more efficient

In this work, we performed an experimental study to verify whether the type of the output size (prime number or power of two) has any real influenc on the performance of non-cryptographic hashes. On the basis of an analysis of tables of collisions and distribution of outputs, it is easy to understand why some authors insist that it is important that M be a prime number.

For example, see Table I, which shows the collisions produced by each NCHF for the dataset LCC. Each row of this table represents the results of an NCHF for each possible hash table configuratio (POWER-SPARSE, POWER-DENSE, PRIME-SPARSE, and PRIME-DENSE). The last column simply identifie the values found in each cell. From top to bottom, these values represent the following:

Table I. Results of the collision test using the key set LCC.

	Pow-Spa	Pow-Den	Pri-Spa	Pri-Den	
FNV-1	254	655	278	653	Total collisions
	1.29777	2.44912	1.33534	2.43833	Average chain length
	4	8	7	9	Longest chain
FNV-1a	245	646	256	648	Total collisions
	1.28422	2.40130	1.30082	2.41176	Average chain length
	4	7	5	8	Longest chain
APartow	241	639	260	655	Total collisions
	1.27829	2.36538	1.30697	2.44912	Average chain length
	4	8	4	8	Longest chain
DJBX33A	238	654	243	660	Total collisions
	1.27388	2.44371	1.28125	2.47651	Average chain length
	4	9	4	7	Longest chain
BuzHash	243	648	246	664	Total collisions
	1.28125	2.41176	1.28571	2.49887	Average chain length
	4	7	4	7	Longest chain
DEK	**425**	721	**267**	657	Total collisions
	1.62317	2.86788	**1.31786**	2.46000	Average chain length
	11	16	**5**	9	Longest chain
BKDR	249	651	264	662	Total collisions
	1.29021	2.42763	1.31317	2.48764	Average chain length
	4	8	5	8	Longest chain
MurmurHash2	251	662	253	651	Total collisions
	1.29322	2.48764	1.29625	2.42763	Average chain length
	4	8	4	7	Longest chain
lookup3	273	663	232	656	Total collisions
	1.32734	2.49324	1.26514	2.45455	Average chain length
	4	9	4	7	Longest chain
SuperFastHash	258	651	250	657	Total collisions
	1.30389	2.42763	1.29172	2.46000	Average chain length
	5	8	4	8	Longest chain

Highlighted in bold are the scores of DEK function for POWER-SPARSE and PRIME-SPARSE. The great difference between them suggests that DEK is not robust with respect to the output size.

1. The total number of collisions that the function produced for the particular configuration
2. The average length of the hashing strings, that is, the average number of keys that have been assigned to each bucket of the table. Buckets that do not hold any key are not considered.
3. The length of the longest hashing chain, that is, the number of keys assigned to the most crowded bucket of the table.

If we consider columns POWER-SPARSE and PRIME-SPARSE, we see that the values are very similar for almost all of the functions. This is the kind of result that a detractor of the *modulo prime myth* would expect, because the load factor and table sizes of both configuration are very similar and thus the total collisions should be very similar as well, if we assume that the use of prime numbers is not really important in practice. However, in the row for the DEK hash function, the values in these two columns (highlighted in bold) are very different. When the size is a power of two, the hash function produces around 60% more collisions, the average length of the hashing chains is 0.31 units greater, and the longest chain grows from 5 elements to 11. This function is competitive when prime numbers are used but suffers from a great reduction in performance with powers of two. These results are consistent with those generated by all of the other metrics for collision resistance and distribution of outputs for the key set LCC.

The DEK function is not the only function that is affected by this problem. Other NCHF show this kind of behavior with particular key sets. Figure 2 summarizes this effect: rectangles represent the collision rate of each NCHF for each key set. Figure 2(a) shows the results when a prime number is used as the output size, and Figure 2(b) gives the results when a power of two is used (both experiments use SPARSE tables). In the firs case, because of the scale we are using, it is difficul to distinguish between the performance of the different functions. However, when we use powers of two and maintain the same scale, it is clear that BKDR, DJBX33A, and both versions of FNV produce an abnormally high collision rate with key sets RANDOM, REPEAT, and SPARSE. Furthermore, function DEK also has problems with key sets PASSWD, SPARSE, and VARIABLE.

All of the other NCHF in this experiment (APartow, BuzHash, MurmurHash2, lookup3, and SuperFastHash) are robust with respect to the table size and do not exhibit abrupt differences in performance between the use of primes and powers of two.

3.3.1.1. Analysis and conclusions. It is worth considering whether these experiments support the modulo prime theory or reinforce the position of its detractors. The use of prime numbers is clearly very important—almost a requisite—for some NCHF that cannot effectively disrupt the internal patterns of some key sets. This seems to support the modulo prime theory. However, one can argue that the principal purpose of a hash function is to smoothly scatter the input data, *dissipate* statistical patterns, and generate apparently random hash values. Moreover, if an NCHF cannot provide this functionality, the proper solution should not be to *patch* this function using a more complex compression map. Instead, this function should be identifie and its use should be discouraged in favor of other functions that can perform normally regardless of the output size.

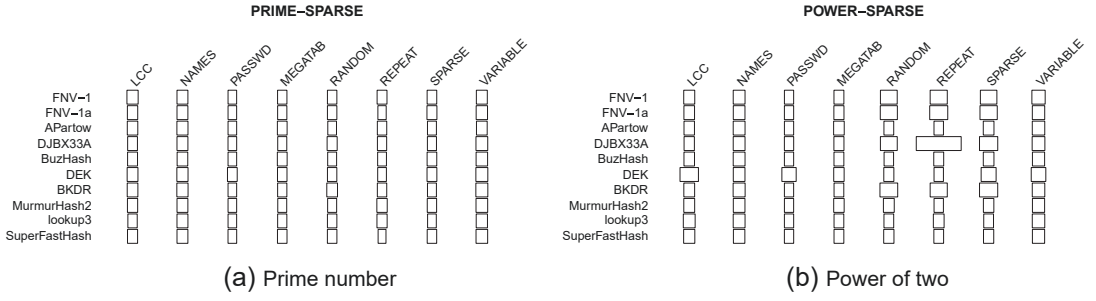

Figure 2. Comparison of the collision rate produced by each function when the output size is a prime number (a) or a power of two (b). Some NCHF that would otherwise be competitive show an important loss of performance with some of the test key sets when powers of two are used.

What is special about functions FNV, DJBX33A, BKDR, and DEK? Why do these particular functions and no others have problems with powers of two? The most plausible explanation is that these NCHF base their mixing functions almost entirely on multiplication, along with other simple operations such as *xor* and summation. These combinations are particularly vulnerable to some patterns that often appear in key sets, so that the outputs of the function sometimes reproduce these patterns. For example, it is easy to consider a mixing function that, under certain circumstances, very frequently produces output values that are multiples of a number x. The problem arises when the size of the table M has a common divisor with x. In this case, the NCHF will always return multiples of that common divisor, thus propagating the error. Because prime numbers do not have divisors (other than 1 and the prime itself), it is very likely that they can *hide* (or counteract) this error. This is why prime numbers are so important for NCHF with weak mixing functions.

3.3.2. Load factor tests. We have already shown that the use of a table size that is a power of two strongly influence the performance of some NCHF. We now want to determine whether the load factor of the table also affects performance. For this purpose, we compared the collision rates produced by functions when using SPARSE and DENSE tables (Figure 3). There are no important differences between plots 3(a) and 3(b). In contrast to the results with the output size, the use of crowded or mostly empty tables did not affect the performance of the NCHF being studied. This result was expected, hash functions are designed to evenly spread their outputs across the whole table, and this should not be related in any way to the number of elements that the table contains at any particular moment. Therefore, all hash functions are expected to work well with different load factors.

The only obvious difference between these plots is that NCHF tend to show more homogeneous performance with DENSE tables. This is a natural effect of the compact population of the tables on the collision rate, which tends to a constant value of $1 - 1/\alpha$ when the load factor (α) increases (see [19]).

3.3.3. Implications of the methodology. In this section, we explain the methods used in the next section. As noted previously, we subjected 10 NCHF to four different tests (avalanche, distribution, collision, and speed), with up to eight different key sets and using four different table configuration (POWER-SPARSE, POWER-DENSE, PRIME-SPARSE, and PRIME-DENSE). Consequently, the results of all of the possible combinations are too extensive for this document. We generated 69 graphs and 25 tables of different dimensions, with a total of 1680 cells. Therefore, we summarize the results and only present those of greatest interest. The remaining results are condensed and summarized only when they provide notable information.

The most important simplificatio involves the results of the output size tests, which show that some functions have problems with output sizes that are powers of two. Once we identifie this weakness, we considered that it would not be interesting to continue to compare these functions with the POWER-DENSE and POWER-SPARSE configurations which would be expected to greatly penalize some of them. We have also shown that there are no important differences in performance

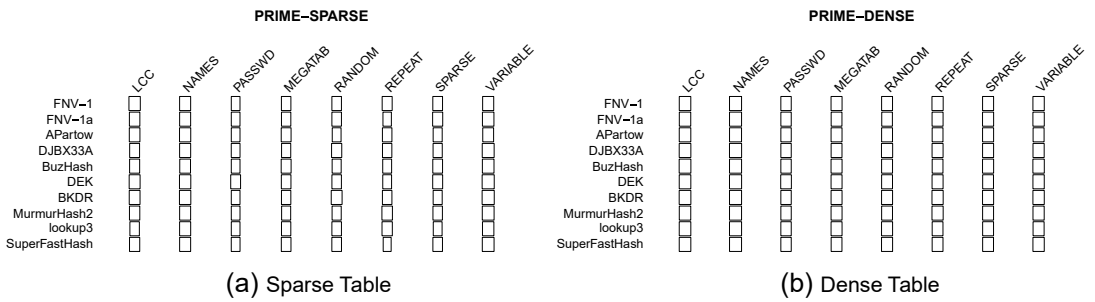

Figure 3. Comparison of the collision rates produced by each function for a sparse table (a) and a dense table (b). There are no important differences in performance with changes in the load factor.

between PRIME-SPARSE and PRIME-DENSE configurations Therefore, for brevity, we will only show the results for one of them. We only show the results with the PRIME-SPARSE configuratio because they have greater variability.

4. EXPERIMENTAL RESULTS

In this section, we compare the 10 most important NCHF in the literature. We seek to analyze them using the most important quality criteria for non-cryptographic hashes, to identify their strengths and weaknesses, and reach conclusions that may be able to aid in choosing the best one for a particular problem.

4.1. Avalanche effect

We begin by analyzing the avalanche effect produced by an NCHF. For this purpose, graphical representations of the avalanche matrices associated with each function are shown in Figure 4. In these representations, which are commonly used in reports on hashing (see, for example, [31] or [13]), the color of the square (i, j) represents the probability that the input bit i affects the output bit j. A red square means that the changes in bit i do not affect bit j at all, or that they affect it 100% of the time ¶. A green square means that bit i has a perfectly random influenc over bit j, so that changes in i cause changes in j 50% of the time.

Two groups can be identifie immediately. Functions lookup3, MurmurHash2, and SuperFastHash show perfect avalanche matrices. Actually, there are three squares in the SuperFastHash matrix that are not completely green, and therefore, the associated values are not exactly 0.5. However, the values are so close (differences on the order of 10^{-3}) that this function can be considered perfect with regard to the avalanche effect.

The avalanche matrix of BuzHash is a borderline case. Although it does not reach the previous avalanche levels, all of the input bits affect the output bits with probabilities that oscillate between 0.4 and 0.6 (approximately), which is a very reasonable, albeit imperfect, approximation.

On the other hand, the functions APartow, DJBX33A, BKDR, and FNV show avalanche levels far from the ideal, with wide red strips in many zones of the matrix. This means that large groups of input bits have no influenc over some output bits, which is a symptom of important weaknesses in an NCHF.

The DEK function represents the most extreme case: the input bits only have two kinds of interaction with the output bits. They either determine the output completely (probability of change equal to 1), or they do not affect them at all (0 probability). In terms of the avalanche effect, DEK is the worst possible function.

4.1.1. Flaws detected by avalanche matrices. Avalanche matrices allow us to immediately spot obvious weaknesses and errors, which would otherwise require a detailed analysis and considerable experience to be detected. The APatrow matrix is a very clear example. It contains a wide vertical red strip on the left side. If we inspect the accumulated probabilities in the avalanche matrix, we fin that the least significan bits of the input $(k_0, k_1, \ldots k_7)$ determine with probability 1 the output bits $(v_0, v_1, \ldots v_7)$ in a one-to-one relationship. Thus, the value of k_0 completely determines v_0, and the same result is seen with k_1 and v_1, k_2 and v_2, and so on. This is not a desirable behavior for an NCHF. Table II offers a practical example of this weakness. It shows the hash values produced by lookup3 and APartow with hashing keys `0x0` and `0x1`. Although lookup3 produces completely distinct results (as we would expect with a high-quality NCHF), APatrow returns in both cases exactly the same hash value, except for the firs bit. The same would happen, for example, with the

¶To achieve a perfect avalanche effect, every input bit must have an apparently random influenc on every output bit. If the state of an input bit completely determines the state of an output bit (100% influence) then there is clearly no appearance of randomness. Thus, 100% influenc is as undesirable as 0%. In fact, the proposed avalanche effect metrics measure the difference in the probabilities of each pair of input/output bits with the ideal probability and assign the same error to a probability of 0 and a probability of 1.

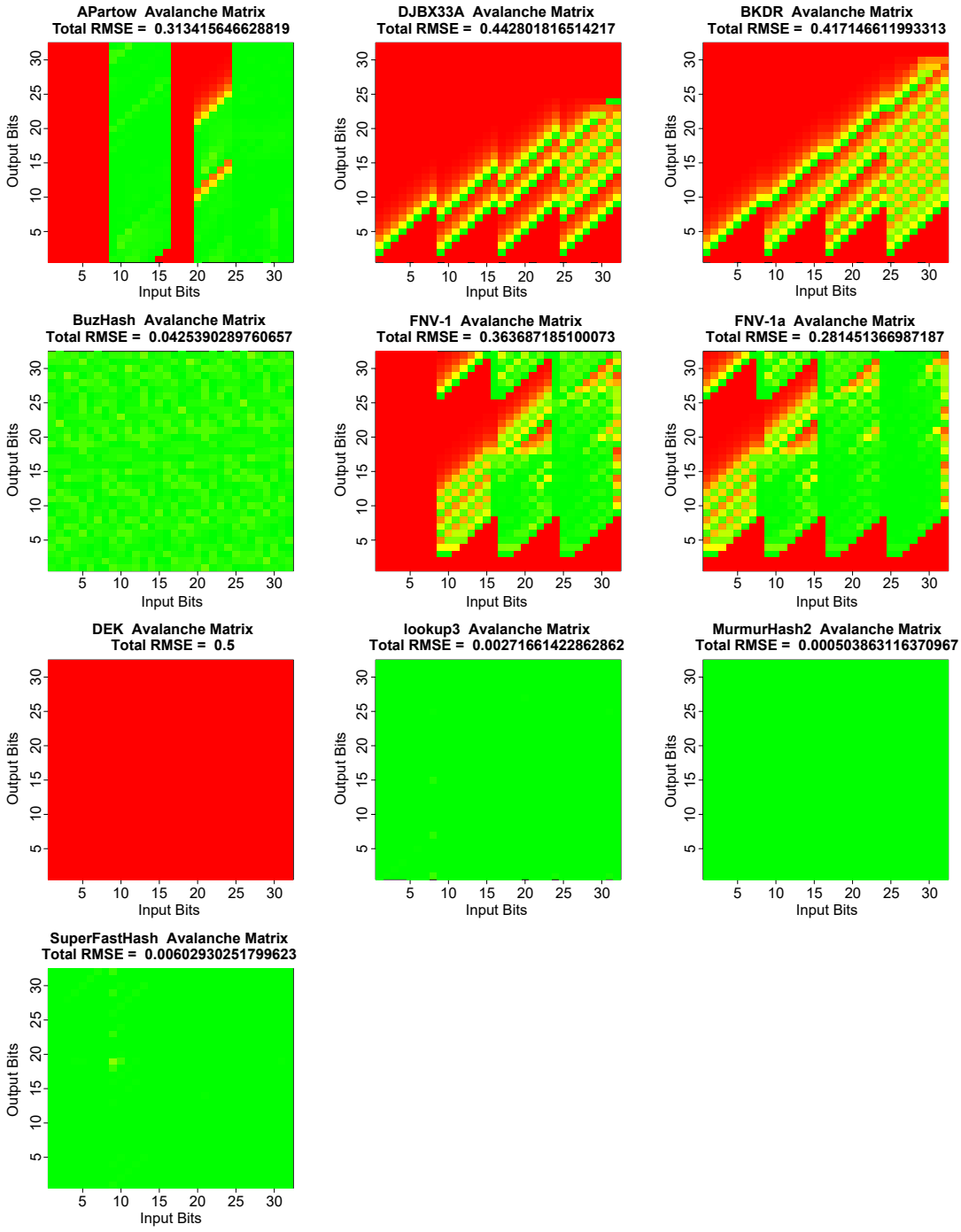

Figure 4. Avalanche matrices of the NCHF studied in this work.

keys `0xFFFFFFF0` and `0xFFFFFFF1`, or with any other pair of keys that differ with respect to only one of the firs 8 bits.

Table III shows an extended version of this experiment and includes all of the NCHF that show wide red areas in their avalanche graphs. Again, lookup3 is used as a control. These results reveal some evident fl ws.

1. For all of the functions. Flipping the least significan bit of input only changes the least significan bit of output.

Table II. Hash values generated by APartow and lookup3 for 2-bit strings that differ with respect to only the least significan bit.

Key	APArtow	lookup3
0x00000000	0xAABD6F8A	0x049396B8
0x00000001	0xAABD6F8B	0x576FAD23

Table III. Hash values generated by low-avalanche NCHF for several bit strings that differ only with respect to the least significan bit.

Key	BKDR	DEK	APartow	FNV-1	DJBX33A	lookup3
0x00000080	0x00000080	0xFFBFFF80	0xAABD6F0A	0xB46A0A95	0x7C5D0F05	0x9184B23A
0x00000000	0x00000000	0x00400000	0xAABD6F8A	0x4B95F515	0x7C5D0F85	0x049396B8
0x00000001	0x00000001	0x00400001	0xAABD6F8B	0x4B95F514	0x7C5D0F86	0x576FAD23
0x00000002	0x00000002	0x00400002	0xAABD6F88	0x4B95F517	0x7C5D0F87	0x83A73853
0xFFFFFFF0	0x226E96C9	0xFFBF800F	0xD205B51B	0x657A2F86	0x7C5C7EB2	0x4022988A
0xFFFFFFF1	0x226E96CA	0xFFBF800E	0xD205B51A	0x657A2F87	0x7C5C7EB3	0x91F79E7D

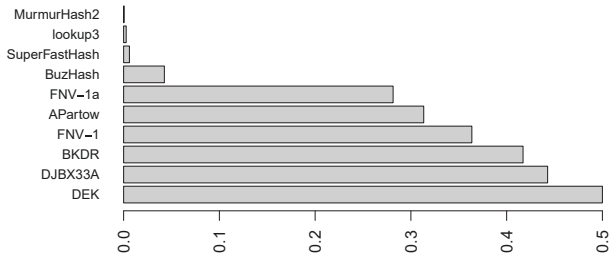

Figure 5. Error (in terms of RMSE) between the avalanche matrix of each NCHF and the ideal matrix.

2. DJBX33A. Inputs `0x0` and `0xFFFFFFF0` produce hash values that are identical with respect to the 15 most significan bits.
3. BKDR. Inputs `0x80`, `0x00`, `0x01`, and `0x02` produce hash values `0x80`, `0x00`, `0x01`, and `0x02`
4. DEK and FNV-1. The eighth bit of input (k_7) has a 100% influenc on all of the output bits from the 8th to the 32nd ($[v_7 \dots v_{31}]$). Flipping k_7 in the input flip all of the output bits $[v_7 \dots v_{31}]$.

4.1.2. Avalanche error. Finally, Figure 5 shows the avalanche matrices in terms of the error (RMSE) between the actual values of the matrix cells and the ideal value 0.5. This bar plot supports the conclusions obtained from the avalanche matrices. That is, MurmurHash2, lookup3, and SuperFastHash have the best avalanche properties, followed by BuzHash and all of the other functions are on a different scale. This graph also sorts the functions according to the avalanche level and thus establishes an avalanche ranking. The best function in this ranking is MurmurHash2, with an error of only 0.0005, followed by lookup3 and SuperFastHash, with errors of 0.0027 and 0.0060. The RMSE of BuzHash is one order of magnitude greater (0.0425), which is still far better than that of FNV-1a (0.2814). The results get progressively worse, until they reach 0.5 (the worst possible avalanche error) for DEK.

4.2. Distribution of outputs

As we explained in Section 3.3.3, for this experiment, we only show the results with the PRIME-SPARSE configuration Table sizes based on powers of two are not considered because some

functions cannot compete when they are used, and the results with the PRIME-DENSE configuration do not contribute any additional or essentially different information.

Figure 6 shows the Bhattacharyya distance between the distribution of outputs generated by each NCHF for each key set and the ideal uniform distribution. Notably, the simpler functions (FNV, BKDR, DJBX33A, and DEK) can compete on an equal footing when compression maps based on prime numbers are used. In fact, the output distribution of APartow is the closest to the ideal with key sets RANDOM and VARIABLE, and FNV and DJBX33A give the second best results with several key sets. On the other hand, lookup3 is the best function with three key sets (LCC, PASSWD, and SPARSE), and BuzHash, MurmurHash2, and SuperFastHash are superior with key sets NAMES, MEGATAB, and REPEAT, respectively.

Interestingly, the differences between functions decrease as the entropy inherent in the key set grows. This occurs because key sets that contain more information are easier to hash. This effect is studied in [35], where the authors deduce that even the simplest NCHF can achieve a very good distribution of outputs due to the fact that the keys themselves sometimes contains great amounts of entropy. In this case, the key sets NAMES and PASSWD are the most entropic, and they produce the most even bar plots.

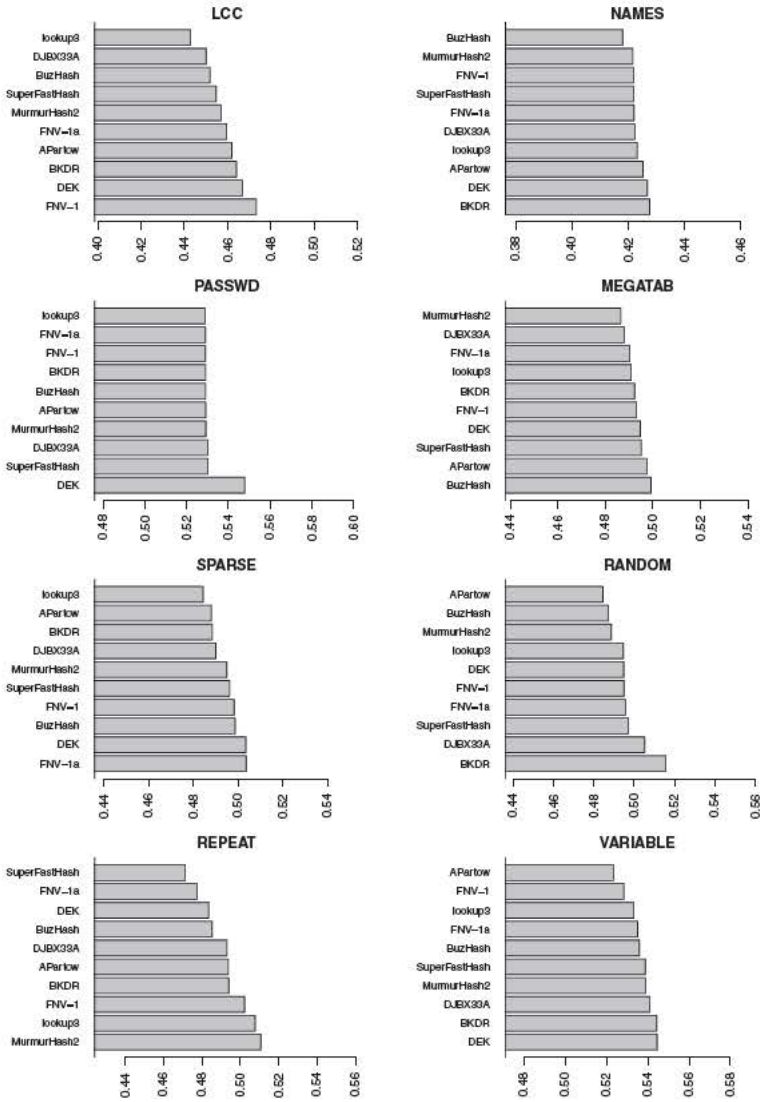

Figure 6. Bhattacharyya distance between the distribution of outputs generated by each NCHF for each key set and the ideal uniform distribution.

4.3. *Collision resistance*

Here, we analyze the results of the collision rate tests using the PRIME-SPARSE configuration, the same as that used in the previous section and for identical reasons (explained in detail in Section 3.3.3). We do not use POWER configurations because of the problems that some functions experience with them and do not use the PRIME-DENSE configuration because it does not add any essentially different information.

Figure 7 shows the collision rate for each NCHF with each key set. The results are consistent with those in the previous section. The functions that give the best distribution of outputs with each key set are the same as those that produce the fewest collisions with that key set. This result was expected, because the distribution of outputs is a more complete criterion than the collision rate. Hash functions with smooth distributions of outputs also have competitive collision rates (whereas the opposite is not necessarily true).

4.4. *Speed*

Figure 8 shows the average time required (ms) by the 10 analyzed functions to hash a database with 10^3 randomly generated keys 1000 times. We repeated the same experiments with different

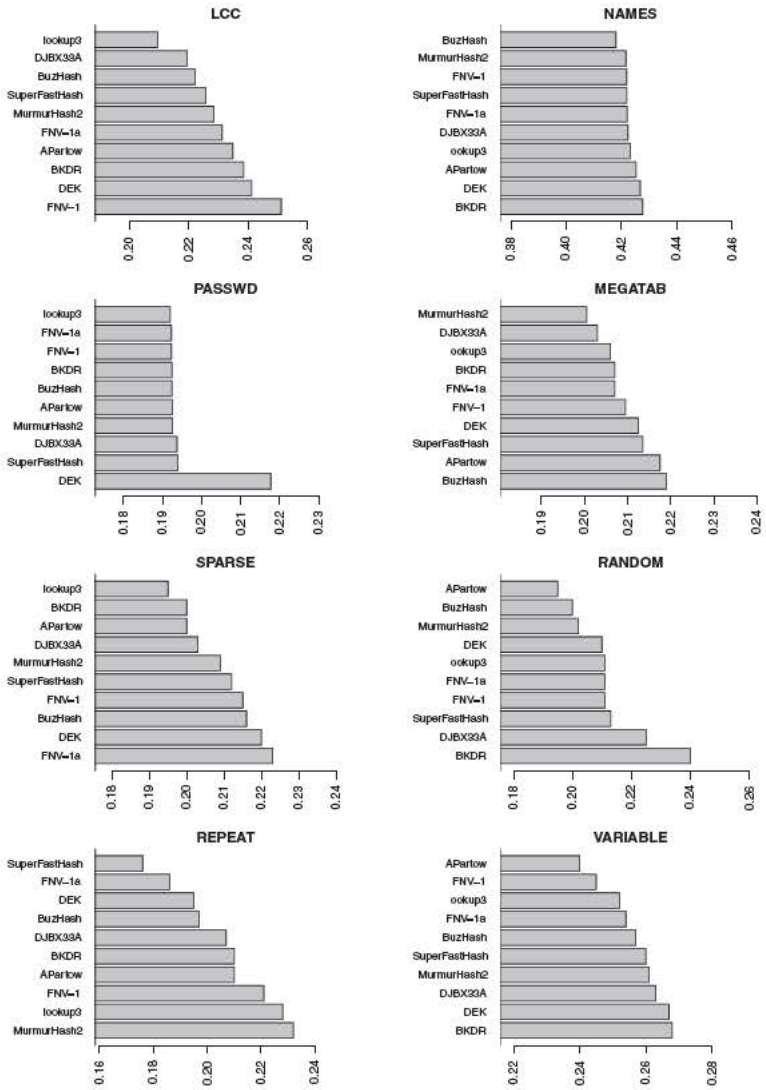

Figure 7. Collision rate for each NCHF with each key set.

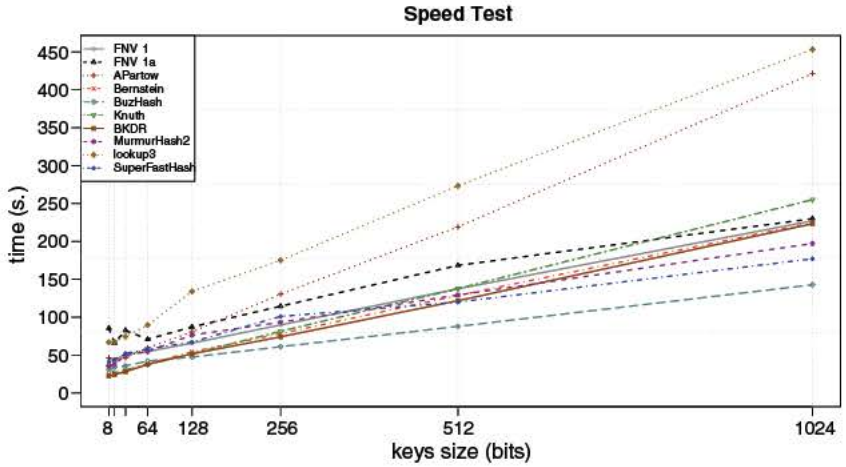

Figure 8. Results of the speed tests.

Table IV. Results of the speed tests.

	8	16	32	64	128	256	512	1024
FNV-1	44,853	42,912	51,424	54,669	65,719	89,762	137,456	227,129
FNV-1a	85,491	65,895	82,662	70,833	87,156	114,475	168,396	229,992
APartow	46,241	41,835	46,688	59,511	80,622	130,454	219,124	421,579
DJBX33A	24,392	25,502	30,150	38,615	54,256	78,216	128,651	224,651
BuzHash	30,841	33,782	35,635	42,152	47,383	61,033	87,800	142,718
DEK	34,519	26,070	29,486	37,282	51,736	81,297	137,682	255,112
BKDR	22,482	24,119	28,042	37,711	51,611	74,070	121,747	223,303
MurmurHash2	35,666	37,889	49,679	54,648	76,364	93,361	129,247	197,256
lookup3	66,881	67,743	74,553	89,590	133,966	175,065	273,301	453,370
SuperFastHash	40,676	44,231	52,027	57,912	67,156	101,045	119,943	177,066

key sizes. The speed of an NCHF also greatly depends on the size of the hashed keys, because of the latency of the function. The x-axis shows the size of the hashed keys, and the y-axis shows the time required by each function to hash the corresponding data (averages of 1000 iterations). These tests were performed on an Intel Core 2 Duo 2.8 GHz with 4 GB of RAM and Mac OS X Snow Leopard as the operating system. Table IV shows the same results as Figure 8 but in numeric format.

As expected, the different functions show different behaviors when the hashed key sets grow. This occurs because some functions have initialization or finalization blocks outside the main mixing block, which gives them a particular latency. Although this latency undermines the overall performance of the function when short keys are hashed, the effect of the latency gradually fades with longer keys. This is the case with SuperFastHash, for example, which uses a finalization block to improve the total avalanche of the output. This is reflected in the graph. Initially, SuperFastHash is among the slowest NCHF but as the keys get longer, the elapsed times for SuperFastHash decrease until eventually it becomes the second fastest function with the longest keys. The opposite pattern is seen with DEK due to its great simplicity, it is very fast with short keys, when other functions are paying a price for their greater complexity. However, with longer keys, it is among the slowest functions.

Surprisingly, lookup3 had rather poor performance, considering its huge popularity in the literature and its extensive practical and industrial applications over the past several years. In contrast, BuzHash, which is a relatively unknown function, gives the best speed scores for keys of 128 bits or larger. In addition to BuzHash, the fastest functions for long keys were SuperFastHash and MurmurHash2. For short keys, the winners were BKDR, DJBX33A, and DEK.

5. DISCUSSION AND CONCLUSIONS

We performed a comparative study of 10 NCHF selected from among the most important in the literature and the software industry: both versions of FNV hash (FNV-1 and FNV-1a), APartow, DJBX33A, BuzHash, DEK, BKDR, MurmurHash2, lookup3, and SuperFastHash. This experimentation was useful for identifying the best functions and for revealing the weaknesses of others.

First, avalanche graphs allowed us to identify a group of NCHF (MurmurHash2, SuperFastHash, and lookup3) that had excellent diffusion properties and gave an almost perfect avalanche effect. Although BuzHash does not reach the same level of excellence as these three, it still shows a very high avalanche effect. Furthermore, we showed that the graphical representation of avalanche matrices can provide, with just a glimpse, a great amount of information on the internal dispersion produced by an NCHF. This makes them a very powerful tool for spotting weaknesses or fl ws that would otherwise be very difficul to detect. For example, with the use of these graphs, we found that a group of functions (APartow, DEK, DJBX33A, BKDR, and FNV) have serious problems when hashing keys that differ only with respect to a specifi set of bits.

The results of data-dependent tests revealed weaknesses in the functions FNV, BKDR, DJBX33A, and DEK. These functions have problems when the compression map uses a power of two as the output size. All of the other NCHF seem to be robust with respect to the output size. This result could support the complaints of many researchers and engineers who have been arguing for the past few years against what they call the *modulo prime myth*. In light of our results, they seem to be right. As long as we choose a sufficientl sophisticated NCHF, it is not necessary to waste time executing a very expensive modulo prime operation. Instead, it is enough to use a logical *and* over a power of two (i.e., select a set of the least significan bits of the produced hash value), which is extremely more efficient On the other hand, the four NCHF that need to use prime numbers are much simpler than the others (in terms of the number of operations they use), so further studies will be needed to determine if substitution of the modulo prime is really worthwhile in terms of total elapsed time.

Interestingly, the functions that needed to use prime numbers also gave the worst avalanche matrices, whereas the functions that generated avalanche graphs with predominantly green squares do not seem to require any particular compression map to operate correctly. This result empirically supports the intuition that the avalanche effect is extremely important, because it greatly influence the ability of an NCHF to dissipate the statistical patterns usually found in key sets, thus generating apparently random outputs, independent of the nature of the hashed keys. Furthermore, the avalanche effect is very easy to calculate and extraordinarily versatile. Avalanche matrices and graphs are perfect for a manual analysis, and these matrices can be trivially converted into error measures (MSE, RMSE, etc.), which are very accurate, continuous, and gradual, and thus perfect for use by an automatic system (such as evolutionary algorithms and other optimization or machine learning techniques).

The reliability of the avalanche effect allows us to distinguish an *elite* NCHF that seems to outperform the others, in terms of the diffusion and pattern-disruption capabilities delivered to the user. This was expected as well, the literature very often claims that lookup3, MurmurHash2, and SuperFastHash are the highest quality NCHF available. However, BuzHash, which is not a very popular function, gave a very good avalanche effect, whereas FNV, which is a very well-known function that has a vast number of important applications, shows two major fl ws: a low-avalanche effect and a lack of robustness with respect to a table size based on powers of two.

Consequently, in light of these results, we recommend that engineers and researchers who need to implement a general purpose NCHF into their systems should choose one of the most advanced functions: lookup3, MurmurHash2 or SuperFastHash. These functions showed no fl ws or weaknesses. BuzzHash also seems to be a good choice. Although it does not give as strong of an avalanche effect as the three aforementioned, it is superior in speed tests with long keys and is still faster than the others with short keys. APartow seems to be robust with respect to the use of powers of two in the compression map, but its avalanche graphs are far from optimal and show a clear weakness for input bits from 0 to 7 and from 16 to 18.

Generally, we cannot recommend the use of the following functions: FNV, DJBX33A, BKDR and DEK. The avalanche tests show that these NCHF do not have good diffusion properties.

Furthermore, they are not robust with respect to the use of powers of two for the compression map. The only advantage of this group of NCHF is their great simplicity, which could offset their shortcomings in some particular applications. In any case, they should be used with care and should always carry out specifi tests in production environments. Among these functions, we would recommend FNV, because it is the most commonly used in the software industry and its avalanche properties are better than those of the others. More precisely, we recommend FNV-1a, which often outperformed FNV-1 in our tests.

Finally, despite the limited reliability of speed tests, it is important for the fina user to consider that some specifi functions, such as BKDR, DJBX33A and DEK, are more suitable for hashing short keys, whereas others, such as BuzHash, SuperFastHash, or MurmurHash2, are better for working with long keys.

ACKNOWLEDGEMENTS

This work was funded by the Spanish Department of Science and Innovation (Ministerio de Ciencia e Innovación) under the research project *Gestión de Movilidad Eficient y Sostenible* (TIN2011-28336).

REFERENCES

1. Vajgel P. Needle in a haystack: efficien storage of billions of photos. Facebook Engineering's Notes, April 2009. Available from: http://www.facebook.com/note.php?note_id=76191543919 [last accessed 17 January 2013].
2. Martin R. The god particle and the grid, April 2004. http://www.wired.com/wired/archive/12.04/grid.html [last accessed 17 January 2013].
3. Monash C. eBay's two enormous data warehouses, April 2009. http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/ [last accessed 17 January 2013].
4. Dean J, Ghemawat S. Mapreduce: simplifie data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113. DOI: http://doi.acm.org/10.1145/1327452.1327492 [last accessed 17 January 2013].
5. Goodrich MT, Tamassia R. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons, Inc.: New York, NY, USA, 2009.
6. Biham E, Dunkelman O. A framework for iterative hash functions-HAIFA. *Second Cryptographic Hash Workshop*, University of California, Santa Barbara, August 24–25, 2006. Available online at: http://csrc.nist.gov/groups/ST/hash/documents/DUNKELMAN_NIST3.pdf.
7. Lucks S. A failure-friendly design principle for hash functions. In *ASIACRYPT, Lecture Notes in Computer Science*, Vol. 3788, Roy BK (ed.). Springer: Berlin Heidelberg, 2005; 474–494.
8. Bertoni G, Daemen J, Peeters M, Van Assche G. Sponge functions. *ECRYPT Hash Workshop 2007* 2007.
9. Bertoni G, Daemen J, Peeters M, Assche GV. On the indifferentiability of the sponge construction. In *EUROCRYPT, Lecture Notes in Computer Science*, Vol. 4965, Smart NP (ed.). Springer: Berlin Heidelberg, 2008; 181–197.
10. Fowler G, Vo P, Noll LC. Fowler / noll / vo (fnv) hash, 1991. Available from: http://isthe.com/chongo/tech/comp/fnv/.
11. Jenkins RJ. Hash functions for hash table lookup. *Dr. Dobb's Journal* 1997. Available from: http://burtleburtle.net/bob/hash/evahash.html [last accessed 17 January 2013].
12. Hsieh P. Hash functions, 2004–2008. Available from: http://www.azillionmonkeys.com/qed/hash.html [last accessed 17 January 2013].
13. Appleby A. Murmurhash 2.0, 2008. Available from: http://code.google.com/p/smhasher/wiki/MurmurHash2 [last accessed 17 January 2013].
14. Klink A, Wälde J. Effective denial of service attacks against web application platforms. Talk at 28th Chaos Communication Congress (28C3), 2011.
15. Crosby SA, Wallach DS. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03, USENIX Association: Berkeley, CA, USA, 2003; 3–3.
16. Knuth DE. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley: Reading, MA, 1973.
17. Partow A. General purpose hash function algorithms, 2010. Available from: http://www.partow.net/programming/hashfunctions/ [last accessed 17 January 2013].
18. Kernighan BW, Ritchie D. *The C Programming Language, Second Edition*. Prentice-Hall: Englewood Cliffs, NJ, 1988.
19. Valloud A. *Hashing in Smalltalk: Theory and Practice*. Lulu: Raleigh, North Carolina, 2008. self-published (www.lulu.com).
20. Henke C, Schmoll C, Zseby T. Empirical evaluation of hash functions for multipoint measurements. *ACM SIGCOMM Computer Communication Review* 2008; **38**(3):39–50. DOI: Available from: http://doi.acm.org/10.1145/1384609.1384614.
21. Knott GD. Hashing functions. *Computer Journal* 1975; **18**(3):265–278.

22. Bellare M, Kohno T. Hash function balance and its impact on birthday attacks. *Advances in Cryptology – EUROCRYPT '04, Lecture Notes in Computer Science*. Springer-Verlag: Interlaken, Switzerland, 2004; 401–418.
23. Sedgewick R. *Algorithms in C++, Third Edition*. Addison Wesley: New Delhi, 2001.
24. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*. The MIT Press: Cambridge, MA, 2001.
25. Uzgalis RC. Hashing concepts and the java programming language. *Technical Report*, The University of Auckland, New Zealand, 1996.
26. Heileman GL. *Data Structures, Algorithms and Object-Oriented Programming*. McGraw-Hill: New York, NY, 1996.
27. Ramakrishna MV, Zobel J. Performance in practice of string hashing functions. In *DASFAA, Advanced Database Research and Development Series*, Vol. 6, Topor RW, Tanaka K (eds), World Scientific Publishing Co. Pte. Ltd.: Singapore, 1997; 215–224.
28. McKenzie BJ, Harries R, Bell T. Selecting a hashing algorithm. *Software-Practice & Experience* February 1990; **20**:209–224. DOI: 10.1002/spe.4380200207.
29. Matsui M, Fukuda S. How to maximize software performance of symmetric primitives on pentium III and 4 processors. In *Fast Software Encryption, Lecture Notes in Computer Science*, Vol. 3557, Gilbert H, Handschuh H (eds). Springer: Berlin / Heidelberg, 2005; 398–412.
30. Bhattacharyya A. On a measure of divergence between two statistical populations define by their probability distributions. *Bulletin of the Calcutta Mathematics Society* 1943; **35**:99–110.
31. Mulvey B. Hash functions, 2007. Available from: http://home.comcast.net/ bretm/hash/ [last accessed 17 January 2013].
32. Fai MTC. General hashing. *PhD Thesis*, University of Auckland, 1996.
33. Uzgalis RC. General hash functions. *Technical Report*, Department of Computer Science University of Hong Kong, Pokfulam Road, Hong Kong, 1992.
34. Fraser C, Hansen D, Hanson D. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, 1995.
35. Mitzenmacher M, Vadhan S. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '08, Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2008; 746–755.