An Evaluation of FNV Non-Cryptographic Hash Functions

Catherine Hayes
Department of Mathematics and Statistics, Maynooth University,
Kildare, Ireland.

David Malone
Hamilton Institute / Department of Mathematics and Statistics, Maynooth University,
Kildare, Ireland.

***Abstract*—We conduct an examination of the FNV family of non-cryptographic hash functions, with comparison to peer functions, across the standard suite of tests combined with commonly-used hash tables. The results obtained raise some interesting questions about evaluation and application of hash functions when considered alongside input type, goals and output distribution. In particular, we review the literature supporting the *Avalanche criterion* for assessing non-cryptographic hashes to understand why it is frequently referenced.**

***Index Terms*—hash functions; hash table; distribution; collisions; avalanche criterion**

I. Introduction to FNV

FNV [1] are a family of non-cryptographic hash functions, of which the current versions are FNV-1 and FNV-1a. The original version of this algorithm was created by Fowler, Noll and Vo in around 1991. The hash family is currently the subject of a draft RFC by the Internet Engineering Taskforce [2]. They were designed to be fast to run, while keeping collision rates low. They have extensive real-world uses including in DNS servers, the Twitter platform, database indexing hashes, major web search / indexing engines, anti-spam filters and more. One search we conducted on GitHub in 2023 found 215,669 instances of a constant associated with the 32-bit version FNV appearing in hosted code. Given this widespread use of FNV, we are interested in evaluating its behaviour, particularly in comparison to hash functions of a similar class.

FNV-1 has a simple structure:

- The hash value, h, is initialised to a constant, known as the "FNV Offset Basis", acting as an Initialisation Vector.
- For each byte of input, b_i, h is multiplied by the "FNV Prime" and then XORed with the input byte b_i.

FNV-1a has the same structure, but with the multiplication and XOR steps done in reverse order. For a given bit-size, the FNV specification describes both how the FNV Prime and Offset Basis are chosen.

In this paper we are going to test the performance of FNV on a number of data sets in comparison to a selection of other non-cryptographic hash functions in a similar class, particularly with respect to collisions arising in hash tables with different data sets.

In Section II, we will describe the test framework used to assess the hashes, including input data sets. In Section III we describe the metrics and performance criteria that we will use. The results of our tests are laid out in Section IV and then discussed in more detail in Section V. We summarise our conclusions in Section VI.

979-8-3503-5298-6/24/$31.00 ©2024 IEEE

II. Test Framework

In order to test the performance of the FNV hash functions, we need to design a framework for testing, and choose some peer hash functions to which we can compare. As one of the most common uses of non-cryptographic hash functions, our chosen input data will be used as keys for a *hash table*, which will distribute the resulting hash outputs into "buckets". We therefore also need to consider appropriate hash table sizes.

A. Input Data

First, we will need input data to be hashed. We choose 4 different styles of data in order to test various aspects of the hash functions' performance:

1) **Real Text: "Baby Names"**
   The top 1,000 Irish baby names for 2021, as per the Central Statistics Office. Each input is just one name, with lengths ranging from 3 to 11 bytes.
2) **Synthetic Text: "Common Words"**
   One thousand distinct strings of varying length each composed of 20 of the most commonly used English words.
3) **Real Bitstring: "IP Addresses"**
   One thousand 32-bit strings consisting of unique IPv4 addresses that had accessed a server during 2015, in chronological order.
4) **Synthetic Bitstring: "Bias"**
   All strings comprising 999 `0xfe` bytes (11111110 in binary) with one `0xff` byte that moves along the input string by one place each time. This results in 1000 distinct inputs heavily biased towards 1-bits, with significant repeating patterns.

Note that words and IP addresses were considered as possible inputs when designing FNV. Repetitive strings differing in only a small number of bits can be difficult for hashes, so our synthetic bitstring input provides a challenging test.

2024 35th Irish Signals and Systems Conference (ISSC) | 979-8-3503-5298-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSC61953.2024.10603139

B. Chosen Hash Functions

We will hash this data using 4 different functions to calculate a 32-bit hash.

1) We will use **FNV-1a** as this is the generally recommended version of FNV.
2) We will also test **FNV-1**.
3) As a comparison, we will test **DJBX33A** [3], one of the simplest hash functions available. This function was created by Daniel J. Bernstein and the name stands for "Daniel J. Bernstein, Times 33 with Addition". As the name would suggest, the hash simply takes the previous hash value, multiplies it by 33 and then adds in the input byte.
4) Finally, the **Murmur2** [4] 32-bit hash, created by Austin Appleby in 2008, is widely used and well known for its avalanche properties[1]. Murmur2 pulls in 4 bytes of input at a time, which are then multiplied by another 32-bit constant, m, XORed with a shift by an integer, and then multiplied by m again. A 32-bit initialisation vector is then combined with this data using multiplication and XOR again. After the final bytes are read in, there is a further mixing step with XOR, shifts and multiplication.

C. Hash Tables

Once the input data has been hashed, it will be distributed and stored in a hash table, by assigning each hashed output to one of a number of buckets via modulo arithmetic, i.e. the bucket chosen is $h \mod M$ where h is our hash output and M is the number of buckets.

Each of our data sets comprises 1,000 lines of input. We decided to use a Load Factor of approximately 2, meaning that we would distribute the output among approximately 500 buckets. More precisely, we wished to examine any potential difference in performance which may arise depending upon the chosen number of buckets. Common recommendations include that M should be either a power of two or a prime number [5]. Since hashes are usually calculated, at least implicitly, modulo a power of two is is unclear if the mixing effect of assigning modulo a prime number is more valuable than the simplicity of selecting a number of output bits by working modulo power-of-two.

We will therefore consider each of the following number of buckets: 500 buckets (to achieve a load factor of exactly 2), 499 buckets (to measure any benefit of using a prime number), and 512 buckets (to measure the performance when combined with a power-of-two number of buckets). The variation in load factor is less than 0.05, so we expect minimal performance changes to arise from the change in load.

III. Performance Criteria

Evaluation of the speed of a hash function is a common performance criterion and has been studied in other works (e.g. [6], [7]). Here, we wished to focus on the quality of the hash output, to examine collisions, study why certain patterns are seen and consider the output distribution. Hence we focus on the following measures of hash performance [8].

[1]We will discuss the Avalanche criterion in Section III-C.

A. Collisions

As we have chosen a load factor of more than 1, we can be confident that we will see collisions in our hash table, i.e. more than one output being mapped to the same bucket. These collisions will be managed using *Separate Chaining*, whereby any colliding entries are chained together into a single linked list of key-value pairs which can then be searched. Naturally, a smaller number of collisions is better, as it means there is less searching required. We measure both the number of buckets with two entries or more and the number of empty buckets as summary indicators of the collisions.

If we consider the process of assigning input to buckets to be comparable with balls being thrown randomly and independently into bins, we can use the Poisson distribution [9] to predict what we might see. A Poisson Distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval, if these events occur independently, with a known constant mean rate. In the case of 500 buckets, for example, our expected mean rate with 1,000 input lines is 2.

We can use this to estimate the probability of each bucket ending up with k inputs, where $k = 0, 1, 2, ...$, and find,

$$\mathbf{P}(X = 0) = 0.135$$

$$\mathbf{P}(X = 1) = 0.27$$

$$\mathbf{P}(X = 2) = 0.27$$

$$\mathbf{P}(X \geq 2) = 1 - \mathbf{P}(X = 0) - \mathbf{P}(X = 1) = 0.59$$

Based on 500 buckets, we therefore expect 68 buckets (13.5%) to be empty, 135 buckets (27%) to have one entry, and 297 buckets (59%) to have collisions of two or more entries.

B. Distribution

It is possible to show that distributing the output uniformly is optimal for hash tables in a number of senses (e.g. average search time [8]). Thus, a statistical measure of the performance of the hash function would be to assess how uniformly the outputs are distributed among the available buckets. We assess this using the chi-squared test, comparing the distribution of outputs amongst the buckets of a hash table to a uniform distribution.

$$\chi^2 = \frac{\sum (O_i - E_i)^2}{E_i},$$

where O_i = observed number of outputs in bucket i and E_i = expected number of outputs in bucket i. We may also convert this χ^2 statistic to a p-value.

C. Avalanche

A good avalanche effect means that for every change to an input bit i, every output bit j should change with 50% probability. The rationale is that if each hash output was assigned independently and randomly of other outputs, then this criterion would hold. This is mooted as being important to create a more random-looking output where similar inputs are not mapped to similar outputs. This is assessed by flipping input bits and noting how often each output bit changes [7].

The avalanche effect differs from Distribution and Collision Resistance in that it is traditionally measured independently of the underlying data. This makes sense as the measure does not particularly care about the actual output, rather it focuses on how the output *changes* when input bits are changed. We will therefore use a random 4-byte input when measuring this metric.

Based on the work done by Bret Mulvey [10], we build an avalanche matrix for the hash function. Each entry of this $(i \times j)$ matrix will be visualised using colour:

- Green $\implies$ toggling input bit i resultes in a change to output bit j approximately 50% of the time. This is the ideal.
- Red $\implies$ the effect of toggling input bit i changes output bit j either 0% or 100% of the time. This is the worst case scenario.
- Yellow $\implies$ a change roughly 25% or 75% of the time.

The statistics were gathered over 10,000 trials of randomly selected inputs.

IV. Test Results

A. Collisions

The level of collisions measured by the number of buckets seeing more than one entry are shown in Figure 1. Each line represents one of our input data sets, with each point showing the result for one of the hash functions.

First, looking at the graph of 500 buckets, we can see that three of the data sets (*Baby Names, Common Words and IP Addresses*) have quite similar results with collisions averaging around 300, and staying within a range of 15 either side of this. Note, this corresponds well with our predicted number of around 297 buckets seeing collisions. However, the *Bias* data set shows a lower number of colliding buckets, especially when combined with the DJBX33A hash function. A similar, but even more pronounced, pattern is apparent when we distribute the hash outputs among 512 buckets. However, when we use a prime number, 499, all data sets performed equally well.

While one might expect a lower number of collisions to be a good thing, it needs to be considered in conjunction with the number of buckets that have been left empty; see Figure 2. The majority of the results cluster around 65 empty buckets, again corresponding well with our predicted number of empty buckets. As before, the *Bias* data set again shows some significant outliers. Again, using 499 buckets produces the most consistent output.

B. Distribution

As described in Section III-B, we used the chi-squared test to measure the distribution of our functions within the hash tables. In Figure 3, the distribution is assessed by the chi-squared p-value, which can be thought of as the probability of obtaining a chi-squared value as large as observed if the null hypothesis were true, i.e. the probability of obtaining these results assuming the assignments to buckets were uniform.

As p-values are a probability, values will be between zero and one. A high probability is supportive of the null hypothesis, which is that the output follows a uniform distribution.

We can see that there is far more variation in results than observed for Collisions and Empty Buckets. However, the *Bias* data set continues to cause most of the problems. For example, we see zero p-values for each of FNV-1a, FNV-1 and DJBX33A when combined with either 500 or 512 buckets.

If we omit the *Bias* data set from our results and look at the distribution of the remaining p-values, we find that they are distributed as expected. Thus, we fail to reject the null hypothesis for each of these data sets. However, we can safely say that the outputs from the *Bias* data set do not appear to be uniformly distributed.

C. Avalanche

Figure 4 shows the avalanche results for our hash functions. Remember that this is based on changes to random inputs, rather than a specific input data set. What is immediately apparent is the excellent performance of Murmur2, especially when compared to the poor performance of DJBX33A.

As briefly described in Section II-B, Murmur2 applies a number of "mixing steps" (multiplication, XOR and shifts) to the data before it is ever combined with the hash value. This creates an input that appears more randomised, which ties in with what the Avalanche criterion is trying to achieve. In fact, if we were to remove the input mixing steps from the Murmur2 algorithm, its avalanche graph would look quite different: see Figure 5. This opens the possibility of taking, say, our *Bias* data set and applying the inverse of mixing to it, resulting in a set of inputs that offer challenges to Murmur2.

V. Discussion

A. Number of Buckets

As we mentioned, a long-standing question concerning non-cryptographic hashes and hash tables is whether the table size should be a prime number or a power-of-two. Supporters of the prime number argument say that a prime number will ensure better distribution, whereas those who prefer to use a power of two argue for its greater speed and efficiency; the operation $\mod 2^x$ can be ignored and one would instead just deal with the least significant x bits of the output.

Our graphs in Section IV-A and IV-B indicate that, for the majority of input sets, the number of buckets is largely irrelevant to distribution and collision. However, for trickier inputs, such as the *Bias* data set, the hashes all performed better when combined with 499 buckets. But then we questioned: Is this

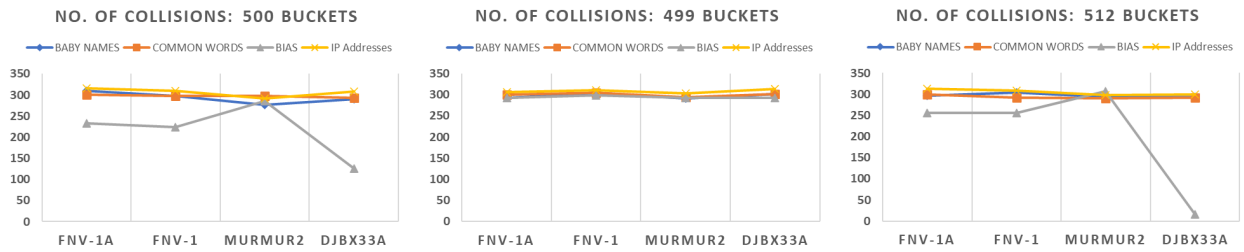


Fig. 1. Number of Collisions Observed for 500, 499 and 512 buckets.

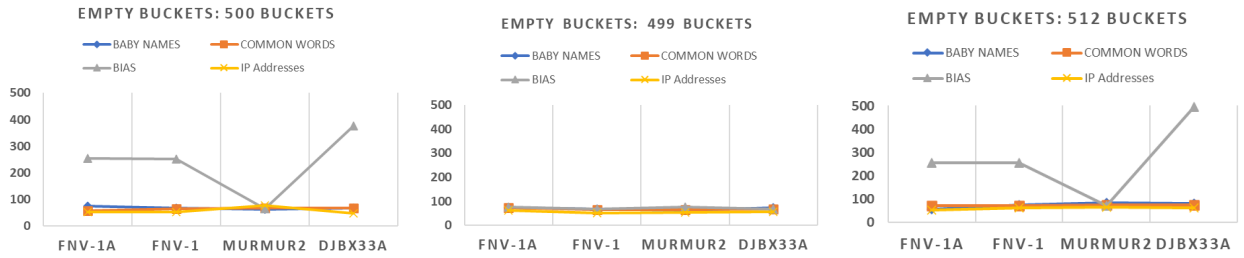


Fig. 2. Empty Buckets Observed for 500, 499 and 512 buckets.

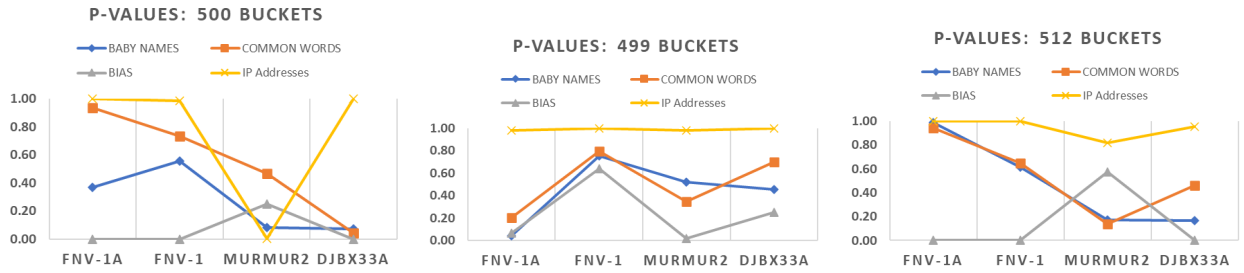


Fig. 3. P-values for the χ^2 test comparing the resulting distribution to uniform for 500, 499 and 512 buckets.

due to the fact that 499 is a prime number or, in fact, is it simply that it is odd?

To check, we looked at an expanded sample set of all numbers of buckets between 488 and 522. Using the FNV-1a hash combined with the *Bias* data set, we examined the number of empty buckets found, along with the p-values for each. See Figure 6 for results.

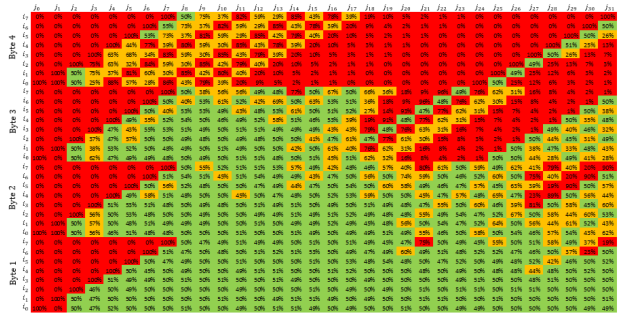
The number of empty buckets (blue line, axis on right hand side) bounces very regularly between a minimum of 60 and a maximum of 271. The Poisson Model would suggest that the number of empty buckets should range from 66 (for 488 total buckets) to 75 (for 522 total buckets). In general, however, when the number of buckets is an even number, only even-numbered buckets are filled, meaning that half are left empty. Therefore using an even number of buckets results in between 227 and 271 buckets being left empty, while an odd number of buckets sees much better results of 60 to 81 empty buckets.

P-values (grey line, axis on left) tell a far more interesting story. What is fascinating is that the highest p-values (i.e. the most supportive of the null hypothesis of uniform distribution) are observed when using 507 or 513 buckets, both of which are composite numbers. Furthermore, two of the lowest p-values are seen when using 499 and 509 buckets, both of which are prime numbers!

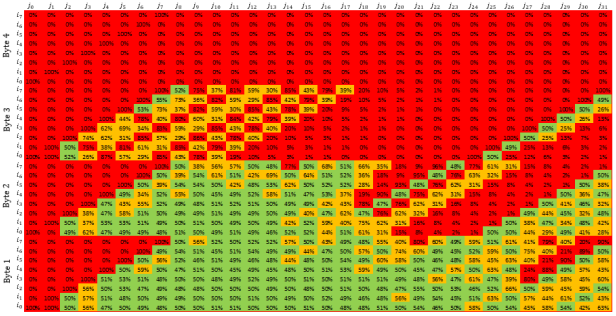
Overall, the average p-value observed for prime numbers (circled in red) in the sample was 0.38, while the average for composite odd numbers was 0.42. Although we had wondered if composite odd numbers might perform as well as primes, we had not expected to see them outperform. Perhaps, after all this time, the debate over prime number versus power-of-two could be reduced to a simple even versus odd question.

B. Empty Buckets for DJBX33A

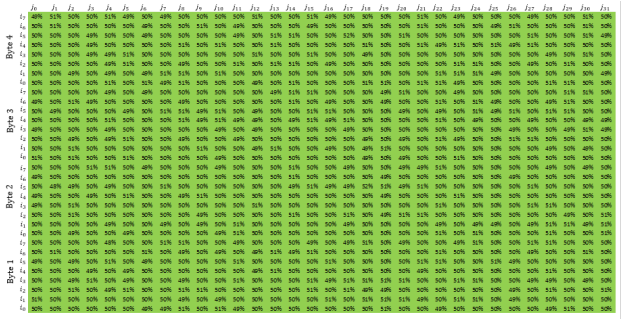
As the most obviously troubling / interesting result, let's take a look at the performance of DJBX33A when combined with the *Bias* data set distributed among 512 buckets, to understand why the poor distrubution occurs. This combination results in just 16 collisions, while there are 496 buckets left empty. Therefore, all of the hash function outputs are being mapped only to these 16 buckets, sending 62/63 results to each and leaving gaps of 32 empty buckets between each filled one.



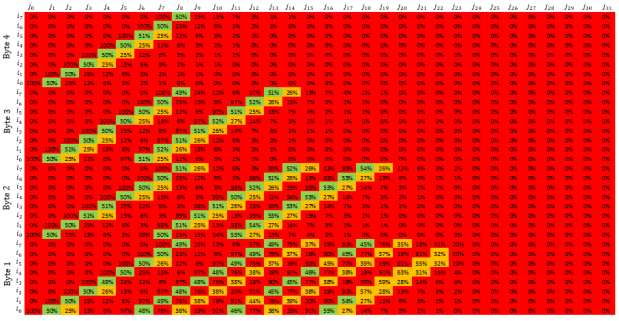
FNV-1a



FNV-1



Murmur2



DJBX33A

Fig. 4. Avalanche 32x32 matrix for each hash function.

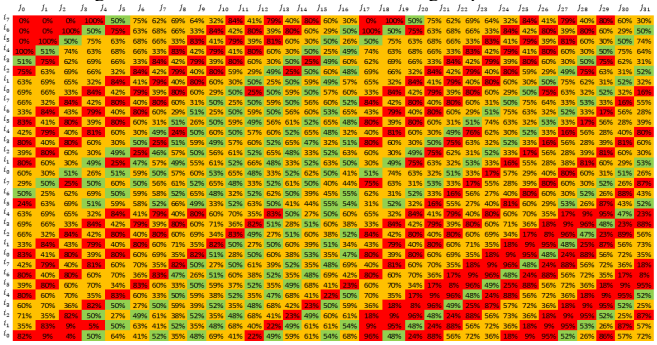


Fig. 5. Murmur2 Avalanche with mixing step removed.

What is causing this pattern? When we consider the mechanics of DJBX33A, the main operation is multiplication by 33. Multiplying a byte of input by 33 is the equivalent of shifting it 5 places to the left, and adding the original byte (33 in binary is 00100001). This means that the bottom 5 bits are preserved through each run of the algorithm.

In the *Bias* data set, every byte of input is identical apart from one. So everything below the shifted byte is simply the addition of the final 5 bits of 999 `fe`'s and 1 `ff` for every input line. Hence the bottom 5 bits will be identical for each hash, which is why only every 32nd (or 2^5) bucket is filled. This is an example of where a prime number of buckets would improve performance dramatically. The lack of a common denominator between 499 and 32 has meant that the pattern is not maintained into the hash table distribution when using 499 buckets. Similarly, we can see that when 500 buckets are used, we have 125 collisions combined with 375 empty buckets: hitting every fourth bucket, as gcd(500,32) = 4.

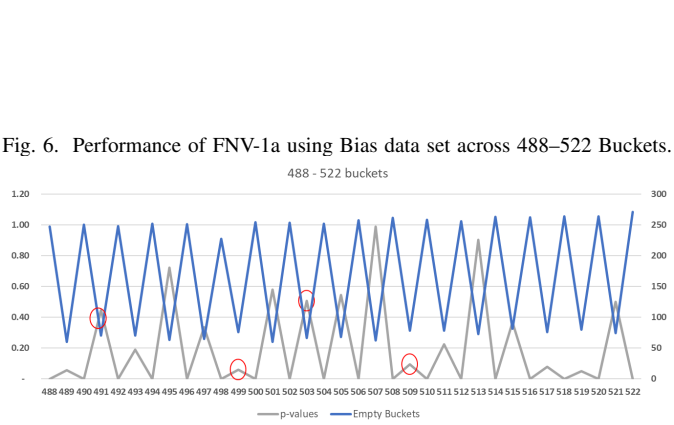


Fig. 6. Performance of FNV-1a using Bias data set across 488–522 Buckets.

C. Uniform Distribution of IP address Outputs

As an interesting side-note, note the consistently high p-values achieved for the *IP Addresses* data set, particularly when combined with FNV-1a, FNV-1 and DJBX33A. This was an unexpected result and one which warranted further exploration. When examining the data set more closely, we noticed quite a high frequency of sequential inputs.

When examining these IP addresses, it became clear that these accesses were, in fact, web crawlers, probably a cluster of machines. Could this sequential nature of the input have an impact on the hash output? Even without the example of

TABLE I
EXAMPLE OF SEQUENTIAL INPUT LINES FROM IP ADDRESSES DATA SET.

IP Address	FNV-1a Hash	Murmur2 Hash
220.181.108.80	0xe49a38c6	0x162466a3
220.181.108.81	0xe59a3a59	0x8e1d95f7
220.181.108.82	0xe29a35a0	0x527f6193
220.181.108.83	0xe39a3733	0x845d8035
220.181.108.84	0xe89a3f12	0x5dc18a76
220.181.108.85	0xe99a40a5	0x04774fe3
220.181.108.86	0xe69a3bec	0xbc9bd96c
220.181.108.87	0xe79a3d7f	0x98c35aa7
220.181.108.88	0xec9a455e	0xffa835df
220.181.108.89	0xed9a46f1	0x2f46efde
220.181.108.90	0xea9a4238	0x10e3332b
220.181.108.91	0xeb9a43cb	0x8c6beb04
220.181.108.92	0xf09a4baa	0x0cfb95d6
220.181.108.93	0xf19a4d3d	0x1b446648
220.181.108.94	0xee9a4884	0xcf9bf437
220.181.108.95	0xef9a4a17	0xb95ebde4

web crawlers, the structure of IP addresses would mean that it could indeed be common to have the higher bits repeating regularly. These inputs would represent a real-world example of where hash functions need to deal with inputs differing by only a small number of bits.

Assuming FNV-1a, the intermediate hash value at each step can be described as follows:

$$h_{i+1} = (h_i \wedge b_i) * p$$

where h = intermediate hash value, b_i = the i^{th} byte and p = FNV Prime. The operations are XOR ($\wedge$) and integer multiplication modulo 2^{32} ($*$). Following the four steps, the final hash output, h_4 can be represented as

$$h_4 = (h_3 \wedge b_4) * p$$

Each of these operations are invertible, hence we can say that

$$h_3 = (h_4 * p^{-1}) \wedge b_4$$

We know from set theory that a function is invertible iff it is bijective, and if a function f is a bijection $f : X \to Y$, then for each element of X there is exactly one element of Y. Considering that the initialisation vector and the first three input bytes were all identical, then h_3 (as the intermediate hash value following the input of the 3rd byte) will show identical output for each line.

If we know that h_3 is identical for each line, the only way that h_4 could also be identical is if b_4, the fourth input byte, was non-distinct. As we know that each of b_4 are distinct, each h_4 as the final hash output must be too. Additionally, the sequential pattern of the input is not well dispersed by the FNV-1a hash. In each sequence, input lines ending in odd numbers correspond to hash outputs ending in odd numbers. Hence, as each sequence switches between even and odd on each line, the output does too, hence improving distribution.

D. Murmur2, Avalanche and Background Literature

In Figure 4 we clearly saw that Murmur2 has superior avalanche properties compared to the other hashes that we considered. However, is it interesting that there was no clear indication in our earlier tests of a notably stronger performance either in terms of collisions or distribution. In fact, there is very little to choose between any of our functions when distribution and collisions are measured using the more "real life" data sets (i.e. excluding *Bias*). Why, therefore, do the avalanche results paint such a different picture?

If we know that the Avalanche criterion is targeting randomisation, is it actually a relevant metric when considered in terms of hash tables and other non-cryptographic uses? The importance of a good avalanche effect is more obvious for cryptographic hash functions. As an example, refer to Table I showing the hash outputs of some of the sequential input lines from the *IP Addresses* data set. The repeating patterns in the FNV-1a hash would make it relatively easy for an attacker to gain some knowledge of the input, whereas the Murmur2 hash outputs are much more random and so potentially relatively more secure.

However, the similarity of performance when measuring distribution and collisions caused us to question if avalanche is actually a good indicator of high quality non-cryptographic hash functions, or has it migrated from the cryptographic world, and just been generally accepted without critical assessment? We reviewed the literature to find where the Avalanche criterion for non-cryptographic hashes may have originated.

1) The paper *"Novel Non-cryptographic Hash Functions for Networking and Security Applications on FPGA"* [11] states that *"Non-cryptographic hashes for applications such as Bloom filters, hash tables, and sketches must be fast, uniformly distributed and must have excellent avalanche properties"* They cite two papers in support of this ([12] & [13]) but, strangely, neither paper actually mentions "avalanche".
2) In the paper *"Performance of the most common non-cryptographic hash functions"* [7], the authors state that *"According to the hashing literature, the most important quality criteria for NCHF are collision resistance, distribution of outputs, avalanche effect, and speed"*. They go on to say that: *"Avalanche effect....is very important for NCHF. 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."*
   There are three sources cited for the latter statement.
   a) The first source is a book self-published by Valloud: "*Hashing in Smalltalk: Theory and Practice*" [14], which discusses in detail the process used to assess various hash functions for use in the Smalltalk language. Valloud says the following in relation to Avalanche. *"Although the avalanche test is quite popular, note that . . . it says nothing of the distribution of the actual hash values. In particular,*

it is quite possible to construct hash functions of horrific quality that nevertheless effortlessly achieve general avalanche" and *"This leads to an extremely important conclusion: employing a scheme which produces seemingly random bits does not necessarily imply nor guarantee that such output will be of good quality when used as hash values"*

b) The second source referenced is a paper called *"Empirical Evaluation of Hash Functions for Multipoint Measurements"* [15]. While this paper does emphasise the importance of a good avalanche effect, the paper itself is attempting to find which hash functions are specifically suitable for hash-based packet selection.

c) The third source, *"Hashing Concepts and the Java Programming Language"* [16], does state that *"Two criteria weed out most candidates for a satisfactory Java language library hash function"* . The first of these is given as *"Adequate mixing test: The algorithm must guarantee that a change in any single bit of a key should result in an equally probable change to every bit of the hash value"*.
This is clearly the Avalanche criterion, but possibly before it is commonly called such. However, the author then goes on to make an important distinction and again comes back to a search for a randomised result: *"A proposal for hash search was described by Hans Peter Luhn in an IBM technical memorandum in 1953. What he wanted was a function that would deliberately abuse keys producing practically the equivalent of the mathematical concept of uniformly distributed random variables. Luhn's goal for producing uniform randoms is one approach, but often in computer science the goal of getting a completely even distribution has been substituted for it. The change in goal is significant and leads to two completely different lines of research both of which are commonly called 'hashing'. Creating even distributions can only be done by considering the structure of the keys, so the method can never be 'general'. However creating random uniform distributions can be done without respect to the structure of the key, and so it can be provided as a standard part of a language library. The word 'hashing', as used in this paper, refers only to the goal of producing a uniform random distribution of a key set."*
This presents a very interesting view, albeit one from 1996. It differentiates clearly between two separate aims that can be achieved by hashing: creating a randomised output, or a well distributed output for a set of keys. Again, avalanche is vital only for those in search of a random result.

3) In the third paper we found, entitled *"An Enhanced Non-Cryptographic Hash Function"* [17], it says that *"The most essential features of non-cryptographic hash functions is its % distribution, number of collisions, performance, % avalanche and quality."* and that *"The criteria for optimization is based on the assertion that, with hash functions, there should be equal probability with the generation of each output and a little change in inputs, must result in a huge change in outputs."* The paper referenced in support of this [18] is written by the same authors as of *"Performance of the most common non-cryptographic hash functions"* referenced in point 2 above, with the same supporting sources.

Another early reference (1996) to avalanche is "*Applied Cryptography*" by Bruce Schneier [19]. These early references to avalanche for cryptographic purposes seem to describe a slightly different effect to what we understand today for non-cryptographic hash functions. Here, when describing the DES (Data Encryption Standard) system, Schneier referencing DES's "expansion permutation" says that it "expands the right half of the data, R_i from 32 bits to 48 bits". The cryptographic benefit of this is described as follows: *"By allowing one bit to affect two substitutions, the dependency of the output bits on the input bits spreads faster. This is called an avalanche effect. DES is designed to reach the condition of having every bit of the ciphertext depend on every bit of the plaintext and every bit of the key as quickly as possible"*

Based on the above sample of evidence, it would seem that avalanche is an important criterion, but perhaps only for hashing which specifically targets randomisation rather than distribution. The references that are cited in support of avalanche being an important metric for non-cryptographic hash analysis actually either state the opposite (such as Valloud's book [14]) or only apply to specific requirements (e.g. Robert Uzgalis' paper [16])

VI. Conclusion

In conclusion, we can say that, for the majority of "real world" data sets, the distribution and collision performance of all four hash functions tested are perfectly acceptable. When a difficult data set, such as the *Bias* set, is introduced, then considerations such as number of buckets become important: a prime, or indeed an odd, number of buckets is needed to break the patterns of the input data. Indeed, when combining the *Bias* data set with the FNV-1a hash function, we found that an odd number of buckets actually produced better distribution patterns than a prime number, for a small sample size of buckets.

Of most interest, however, was the fact that the avalanche performance of the various hash functions seemed to have little relation to their collision resistance or distribution. Murmur2's excellent avalanche graph compared to the poor performance of DJBX33A on page 5 does not translate into better distribution in a hash table, which is one of the main uses of non-cryptographic hash functions. This led us to ask if perhaps avalanche is not the correct metric to measure when evaluating non-cryptographic functions. We researched papers which had

supported its use, but found quite a contrasting and more nuanced message within these references.

REFERENCES

[1] L. C. Noll, "FNV hash," http://www.isthe.com/chongo/tech/comp/fnv/.

[2] G. Fowler, L. C. Noll, K.-P. Vo, Donald E. Eastlake 3rd, and T. Hansen, "The FNV non-cryptographic hash algorithm," https://datatracker.ietf.org/doc/draft-eastlake-fnv/.

[3] C. Torek and D. Bernstein, "Open conversation on comp.lang.c between Chris Torek and Dan Bernstein among others." https://groups.google.com/g/comp.lang.c/c/lSKWXiuNOAk, 1991, contributions from C. Torek on 18 Jun 1991 and 21 Jun 1991 with D. Bernstein on 25 Jun 1991.

[4] A. Appleby, "MurmurHash source code for the (slower) endian-neutral implementation," https://github.com/aappleby/smhasher/blob/master/src/MurmurHash2.cpp, 2015.

[5] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1998, ch. 6.4: Hashing.

[6] "Use fast data algorithms," https://jolynch.github.io/posts/use_fast_data_algorithms/, 2021.

[7] C. Estébanez Tascón, Y. Sáez Achaerandio, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," https://e-archivo.uc3m.es/bitstream/handle/10016/30764/performance_JSPE_2014_ps.pdf;jsessionid=1D3EDF84B573A8BD2594268650E9B223?sequence=2, 2014.

[8] C. Hayes, "Non-Cryptographic Hash Functions: Focus on FNV," https://mural.maynoothuniversity.ie/18141/1/Final%20thesis%20submission%20to%20examination%20office%20Jan%202024.pdf, 2023.

[9] M. Mitzenmacher and E. Upfal, *Probability and Computing*. Cambridge University Press, 2017, ch. 5.3 The Poisson Distribution.

[10] B. Mulvey, "The Pluto Scarab," https://web.archive.org/web/20230603152138/https://papa.bretmulvey.com/post/124027987928/hash-functions.

[11] T. Claesen, A. Sateesan, J. Vliegen, and N. Mentens, "Novel non-cryptographic hash functions for networking and security applications on FPGA," https://www.esat.kuleuven.be/cosic/publications/article-3374.pdf, 2021.

[12] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," https://dl.acm.org/doi/pdf/10.1145/362686.362692, 1970.

[13] Graham Cormode and S.Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," https://dsf.berkeley.edu/cs286/papers/countmin-latin2004.pdf, 2005.

[14] A. Valloud, "Hashing in Smalltalk: Theory and practice," self-published (www.lulu.com), 2008.

[15] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for multipoint measurements," http://ccr.sigcomm.org/online/files/p41-v38n3i-henkeA.pdf, 2008.

[16] R. Uzgalis, "Hashing concepts and the Java programming language," http://www.serve.net/buz/hash.adt/java.000.html, 1996.

[17] V. Akoto-Adjepong, M. Asante, and S. Okyere-Gyamfi, "An enhanced non-cryptographic hash function," https://www.ijcaonline.org/archives/volume176/number15/akotoadjepong-2020-ijca-920014.pdf, 2020.

[18] C. ESTEBANEZ, Y. SAEZ, G. RECIO, and P. ISASI, "Automatic design of noncryptographic hash functions using genetic programming," https://e-archivo.uc3m.es/bitstream/handle/10016/30762/automatic_CI_2014_ps.pdf, 2014.

[19] B. Schneier, *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc, 1996, ch. 12.2 Description of DES.