Non-Cryptographic Hash Functions: Focus on FNV

Catherine Hayes

Master of Science (M.Sc.) Thesis

Supervisor: Dr. David Malone
Head of Department: Prof. Stephen Buckley

Department of Mathematics & Statistics
Maynooth University
September 2023

This thesis has been prepared in accordance with the M.Sc. regulations of Maynooth University and is subject to copyright.

Acknowledgements

First, I would like to thank the Maynooth University Department of Mathematics and Statistics, for warmly welcoming an old student back into the fold. It was also very valuable to share thoughts and ideas with my fellow M.Sc. students during our weekly seminars with Dr. Detta Dickinson.

My thanks too to Mr. Landon Curt Noll, co-creator of the FNV Hashes, for so generously sharing his time and knowledge with us, giving us a valuable insight into the history and evolution of the FNV hash functions.

Last, but by no means least, my eternal gratitude to my supervisor, Dr. David Malone. His patience, generosity and enthusiasm made the creation of this thesis a thoroughly enjoyable experience. His love of maths and depth of knowledge were inspiring, and I left every one of our meetings invigorated and excited to learn more.

(*Small mention to my cats, Pythagoras and Minerva, who offered their assistance by sleeping through many hours of research*)

Abstract

In this thesis, we will explore the world of hash functions. After a brief overview of the construction and uses of cryptographic hashes, we will then focus almost exclusively on non-cryptographic functions. We delve into the *FNV* family of hash functions in significant detail, examining their background, structure and motivation. We then introduce some well-known peer functions, against which FNV can be tested. We lay out our test parameters, where we will utilise *hash tables*, a variety of input types, and explain our choice of *load factor, collision management* and more. We run rigorous tests measuring *distribution, collision resistance* and *avalanche effect* which highlight some significant differences in performance. The observed test results are examined in detail, with explanations provided for the varying performances of the functions. These results also provoked some interesting questions around the accepted methodology for measuring the performance of non-cryptographic hash functions, particularly the relevance of the avalanche effect. We examine published works which reference this metric and discover that, perhaps, its usefulness has been overstated. We finally consider collision resistance in a more abstract sense, examining how each function performs when tested with a more challenging input size.

Contents

1 Introduction **7**

2 Hash Functions **9**
2.1 Introduction to Hash Functions . . . 9
2.2 Cryptographic Hash Functions . . . 9
2.3 Non-Cryptographic Hash Functions . . . 14
2.3.1 Hash Tables . . . 15
2.3.2 Collision Management . . . 16
2.3.3 The link between minimizing empty buckets and good distribution . 17
2.3.4 General Testing Methodology . . . 19
2.4 Summary . . . 20

3 The FNV Hash Algorithm Family **21**
3.1 The FNV Hash Functions . . . 21
3.2 Interview . . . 23
3.3 Operators . . . 26
3.4 Summary . . . 28

4 Design of our Test Framework **29**
4.1 Our Chosen Test Methodology . . . 29
4.2 How many collisions *should* we expect? . . . 34
4.3 Summary . . . 35

5 Tests & Results **36**
5.1 Avalanche . . . 36
5.1.1 The Mechanics . . . 36
5.2 Distribution and Collisions . . . 44
5.2.1 Are these results in line with expectations? . . . 49

6 Analysis of Results **51**
6.1 Why do FNV-1, FNV-1a and DJBX33A perform so poorly when using the Bias data set? . . . 51
6.2 What effect does the use of a prime number have on distribution and is the fact that it is prime actually the pivotal factor? . . . 52
6.3 Why are we seeing a disconnect between avalanche performance and distribution? . . . 54
6.4 Can we say with confidence that we see a normal distribution when we look at the distribution of empty buckets? . . . 57
6.4.1 Detailed Statistical Inspection . . . 57

6.5 Why are the results for FNV-1 and FNV-1a identical when using the Bias data-set combined with 512 buckets? . . . 61
6.6 Why do all hashes except Murmur2 achieve nearly perfect p-values on the IP Addresses data set? . . . 61
6.7 Summary . . . 66

7 Observations on Results, Revelations and Arising Questions 67
7.1 Is Avalanche really a relevant metric? . . . 67
7.2 Choice of bucket size depends on algorithm used . . . 70
7.3 Generating Collisions for FNV . . . 71
7.3.1 Theory of Collisions . . . 71
7.3.2 Considering Collisions More Generally . . . 74
7.4 Summary . . . 78

8 Conclusion 79
8.1 My personal conclusions . . . 79
8.2 Discoveries . . . 80
8.3 Further Work . . . 81
8.4 Summary and farewell . . . 81

A Convolution 82

B Avalanche Graphs using Data Set Inputs 83

C Poisson Expectations vs Observed Results for 499 and 512 buckets 96

D DJBX33A Collision Resistance 98

List of Figures

2.1 Merkle-Damgard Construction . . . 10
2.2 Hash Table Structure . . . 15
2.3 Separate Chaining . . . 16

3.1 Shift Operation on 8 bits . . . 27
3.2 Rotate Operation on 8 bits . . . 28

5.1 FNV-1a One byte Avalanche . . . 37
5.2 FNV-1a 32 x 32 Avalanche . . . 38
5.3 Measuring matrices of varying length from the end . . . 38
5.4 Measuring from Byte 1 . . . 39
5.5 FNV-1 32 x 32 Avalanche . . . 39
5.6 DJBX33A One byte Avalanche . . . 40
5.7 DJBX33A 32 x 32 Avalanche . . . 41
5.8 DJBX33A 32 x 32 Avalanche with IV = 5381 . . . 42
5.9 Murmur2 32 x 32 Avalanche . . . 42
5.10 Murmur2 One byte Avalanche . . . 42
5.11 FNV-1a 32 x 64 Avalanche . . . 43

6.1 FNV-1a P-values and Empty Buckets Obverved for 496 - 516 Buckets . . . 54
6.2 Murmur2 with Mixing Step Removed 32 x 32 Avalanche . . . 55
6.3 CDF: Observed results vs normal distribution . . . 58

7.1 Probability of Finding Collisions . . . 72
7.2 Probability of collisions for 64-bit and 128-bit FNV-1a . . . 73

B.1 Avalanche Matrix: FNV-1 with Baby Names DataSet . . . 84
B.2 Avalanche Matrix: FNV-1 with Common Words DataSet . . . 85
B.3 Avalanche Matrix: FNV-1 with Bias DataSet . . . 86
B.4 Avalanche Matrix: FNV-1 with IP Addresses DataSet . . . 86
B.5 Avalanche Matrix: FNV-1a with Baby Names DataSet . . . 87
B.6 Avalanche Matrix: FNV-1a with Common Words DataSet . . . 88
B.7 Avalanche Matrix: FNV-1a with Bias DataSet . . . 89
B.8 Avalanche Matrix: FNV-1a with IP Addresses DataSet . . . 89
B.9 Avalanche Matrix: Murmur2 with Baby Names DataSet . . . 90
B.10 Avalanche Matrix: Murmur2 with Common Words DataSet . . . 91
B.11 Avalanche Matrix: Murmur2 with Bias DataSet . . . 92
B.12 Avalanche Matrix: Murmur2 with IP Addresses DataSet . . . 92
B.13 Avalanche Matrix: DJBX33A with Baby Names DataSet . . . 93
B.14 Avalanche Matrix: DJBX33A with Common Words DataSet . . . 94
B.15 Avalanche Matrix: DJBX33A with Bias DataSet . . . 95

B.16 Avalanche Matrix: DJBX33A with IP Addresses DataSet 95

D.1 Pattern of Chain Lengths for DJBX33A 2-byte inputs 99

List of Tables

3.1 Offset Bases for FNV . . . 22
3.2 FNV Primes . . . 23
3.3 BitwiseAnd: Behaves like $(a \times b)$mod 2 . . . 27
3.4 BitwiseOr: Returns 1 when either one OR both inputs = 1 . . . 27
3.5 BitwiseXOR: Returns 1 when inputs are different . . . 27

5.1 Top 1000 Baby Names in Ireland, 2021: Distribution and Collision Resistance 45
5.2 Permutations of the 1000 Most Common English Words: Distribution and Collision Resistance . . . 46
5.3 Synthetic Bit String Biased (87.5%) Towards 1-bits: Distribution and Collision Resistance . . . 47
5.4 IP addresses: Distribution and Collision Resistance . . . 48
5.5 Summary of results for 500 buckets . . . 49

6.1 FNV-1 with Murmur2's Finalisation Step . . . 56
6.2 Benjamini-Hochberg Procedure Ranking . . . 60
6.3 Distribution and Collision Resistance of UNHASHED IP Addresses . . . 61
6.4 Distribution and Collision Resistance of Random 4-byte Data . . . 62
6.5 Example of Sequential Input lines from IP Addresses Data Set . . . 63
6.6 Distribution and Collision Resistance for Data Set: New IP Addresses from 2023 . . . 64

7.1 Distribution of 32 Outputs mod 6 . . . 70
7.2 Probabilities of collisions by input size . . . 73
7.3 FNV-1 and FNV-1a: Distribution of all possible 4-byte inputs . . . 75
7.4 Finalisation Step Comparisons . . . 76
7.5 Murmur2 & Murmur3: Distribution of all possible 4-byte inputs . . . 76
7.6 DJBX33A: Distribution of all possible 4-byte inputs . . . 76
7.7 Distribution of DJBX33A for all possible 2-,3-, and 4-byte inputs . . . 77
7.8 Distribution of SHA Compared to Poisson Estimates . . . 78

C.1 Summary of results for 499 buckets . . . 96
C.2 Summary of results for 512 buckets . . . 97

D.1 Distribution of DJBX33A for all possible inputs . . . 98

List of Algorithms

1 Pseudocode for 32-bit FNV-1 and FNV-1a algorithms 22
2 Pseudocode for DJBX33A 31
3 Pseudocode for Murmur2 algorithm 32
4 Pseudocode for Avalanche Test 34

Chapter 1

Introduction

This thesis will introduce the reader to the world of hash functions, with the main focus being on non-cryptographic functions. We will compare four well-known algorithms using the traditionally accepted suite of tests, sometimes with surprising conclusions. While some of our results will reinforce commonly-held beliefs about the comparison of hash functions, others will call into question their ongoing validity.

In Chapter 2, we introduce the idea of hash functions, starting with cryptographic functions. Cryptographic hash functions are used where a high level of security is required, such as data integrity and password protection. We will look at how these are typically constructed, some of the most widely used functions and how their level of security is measured. Non-cryptographic hash algorithms are used when efficient storage and search functions are required. Hash tables are one of the most common uses of non-cryptographic functions, and will be used in all of our subsequent tests. We explain how these are structured and managed before briefly introducing the commonly accepted methodology used to test and compare non-cryptographic hash functions.

Chapter 3 will then delve more deeply into one group of hash functions in particular: the FNV hashes. These non-cryptographic hash functions are widely used in familiar applications such as Twitter, video games, and major search engines. We examine their structure in detail, and explain the rationale behind each step. From here, we will look more generally at what mathematical operators are commonly used in hash functions and why. To complete this chapter, we have words direct from one of the authors of FNV himself — my supervisor and I were lucky enough to have a phone interview with Mr. Landon Curt Noll, where we gained valuable insight into the motivation and ideas behind the functions.

In Chapter 4, we will look at the widely accepted suite of tests using for comparing various hash functions. We will set out our own test criteria, such as choosing relevant data sets to be tested, load factor and hash table size. We will also introduce two other functions which will be tested alongside FNV-1 and FNV-1a for comparison purposes: Murmur2 and DJBX33A. Both of these functions are well known and widely used but with very different priorities: Murmur2 is known for its impressive avalanche performance, while DJBX33A is famous for simplicity and speed. Before starting to run the tests, we will take a quick "mathematical sense-check" and ask ourselves how many collisions we should expect to see, having previously (in Chapter 2) also examined why keeping empty buckets to a minimum is so important for a good distribution of results.

In the next chapter, we will see the results of Avalanche, Distribution, and Collision Resistance testing. Starting with Avalanche, and using FNV-1a as the recommended

FNV hash, we work our way up from just one byte of input to four, while examining and interpreting the patterns which appear. We then compare this to the avalanche matrices created by FNV-1, Murmur2 and DJBX33A. We then compare all four functions across a number of metrics to analyse their distribution and collision performance via the number of empty buckets seen, the number of collisions, and the length of collision chains, along with chi-squared distribution results.

Chapter 6 will analyse these results. There are six notable questions which arose out of the tests, all of which will be examined in detail, sometimes with surprising conclusions. We analyse why certain patterns appear and why some functions perform better than others when combined with specific data input or hash table bucket size.

In Chapter 7 we use the findings from the previous chapter, plus some further research, to question some of the long-held beliefs around hash testing. We question the validity of Avalanche testing as a measure of the performance of non-cryptographic functions. We also suggest that the long-standing question of bucket size being either a prime number of power-of-two is neither restricted to just these two options, nor is it a unilateral decision. We then examine the difference between collisions within hash tables and collisions which are the result of two distinct inputs having the same hash outputs. When tested for the latter type of collision, our chosen hash functions do reveal more weaknesses than previously seen.

Finally, in chapter 8 we look back on the work done, what we have learned from it, and where we could pursue further.

Chapter 2

Hash Functions

2.1 Introduction to Hash Functions

A hash function is an algorithm which takes input data of any size and converts it to a deterministic output of specific length. Hashing is a one-way process; unlike encryption, it is not intended to be reversed. The security level of the algorithm varies depending on requirements. For data integrity purposes, digital signatures or password protection, a cryptographic hash function should be used. Non-cryptographic hash functions are widely used in the areas of data storage via hash tables, bloom filters, checksums and caches. We take a brief look at cryptographic functions before delving into non-cryptographic functions in more detail.

2.2 Cryptographic Hash Functions

The main properties of a cryptographic hash function are [1]:

1. Pre-Image Resistance: For a hash output of x, it should be infeasible to find a message m, such that $h(m) = x$, i.e. it should be nearly impossible to reverse the function to find m.

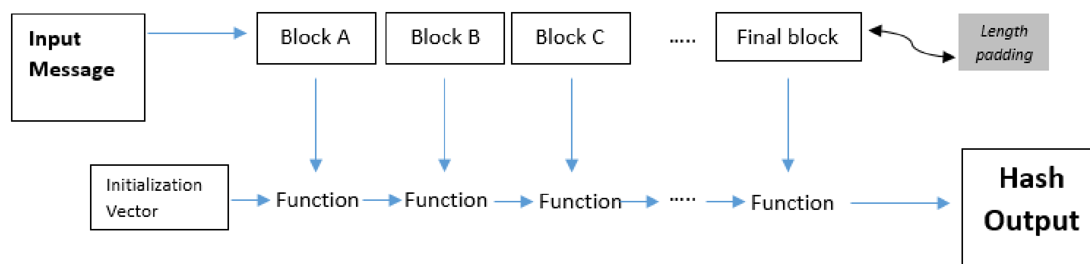
2. Second pre-image resistance: Given a message m_1, it should be hard to find another message m_2 such that $h(m_1) = h(m_2)$

3. Collision Resistance: It must be hard to find any two messages m_1 and m_2 such that $h(m_1) = h(m_2)$

In addition, it is important to attain a good avalanche effect. A good avalanche effect implies that if so much as a comma is changed in the input, the entire hash output should be different. A secure hash algorithm should aim for an avalanche effect of 50%, i.e. a one bit change in the input will effect a change in each bit in the output with 50% probability. The definition of avalanche according to Bret Mulvey [2] (whose code we will use later in Section 5.1 to measure the effect) is as follows:

> "A function is said to satisfy the strict avalanche criterion if, whenever a single input bit is complemented (toggled between 0 and 1), each of the output bits should change with a probability of one half for an arbitrary selection of the remaining input bits."

Figure 2.1: Merkle-Damgard Construction



Construction

The traditional construction for both cryptographic and non-cryptographic hash functions was the **Merkle Damgard Construction** [3]. The Merkle Damgard Construction is a method of building collision-resistant hash functions from collision-resistant one-way compression functions. A one-way compression function takes two fixed-length inputs and produces a single fixed-length output of the same size as one of the inputs. For example, Input A of 128 bits and Input B of 256 bits are compressed together into one output of 128 bits.

The Merkle Damgard Construction takes an input, and breaks it into chunks of equal size (usually 512 or 1024-bit blocks). Without loss of generality, let's assume 512-bit blocks. The message must then be "padded" to make sure each block is precisely 512 bits. This is done by adding 1 plus however many zeros are necessary to bring the length of the message to 64 bits fewer than a multiple of 512. Padding is always done, even if the length of the message is already congruent to 448 modulo 512. Hence, at least one bit and at most 512 bits are appended to the input. The final remaining part is filled with 64 bits representing the length of the original message modulo 64.

The algorithm starts with a fixed initialization vector. It then takes the first block of input data, and combines the two with a compression function, applying a combination of XOR, shifts, modular addition, multiplication and rotation (see section on Operators on page 26) . The resultant hash from the initialization vector and Block 1 then becomes the initialization vector for Block 2, which then runs through each of these rounds of operations again. After all blocks have been combined and mixed, the function produces a final hash of fixed size, generally varying from 128 bits to 512 bits, depending on the algorithm. See Figure 2.1.

Each of these one-way functions require all of the same criteria as cryptographic hash functions (pre-image resistance, collision resistance etc.) so if the one-way compression function is cryptographically secure, then the resultant hash function will be too, provided the message is also correctly padded.

Common Hashing Algorithms

Hashing algorithms are constantly being developed as older versions are "broken" when collisions start to appear, either by attack or investigation. Message Digest (MD) is one family of hashes, starting with MD1, although MD5[4] (developed by Ronald Rivest in 1992) was the standard for many years until collisions were discovered in 2004. MD5 is a

128 bit algorithm which computes 16 functions within the compression function on each block, doing this 4 times before moving to the next block. This is referred to as running 64 rounds.

Secure Hashing Algorithm (SHA)[5] is another family, incorporating SHA-1 which has now been defunct, and the family of SHA-2's which includes the commonly used SHA-256, a 256 bit algorithm, and more recently SHA-512. SHA 256 will run through 64 rounds of these functions, while SHA 512 will compute 80 rounds.

The initialization vector (IV) is required to be an input of the same length as the other block sizes. While traditional encryption systems may require a unique and/or random IV to be used each time, this would not fit with the deterministic nature of hash functions. Therefore, the IV used in hash functions should be somewhat randomly and yet transparently chosen so that they are beyond manipulation by the algorithm designer, and will be used each time that algorithm is run. For that reason, many are chosen based on, for example, the digits of π or e; normally distributed digits and known to everyone. These chosen numbers are known as "nothing-up-my-sleeve numbers" as they show that the programmer did not choose a specific number which may allow them to break or "backdoor" the system themselves; digits from well-known constants are above suspicion. Looking at our two examples above, MD5 used the trigonometric sine function to generate constants for the IV, while SHA-2 uses the square and cube roots of small prime numbers.

Uses of Cryptographic Hash Functions

The criteria for cryptographic hash functions (namely pre-image resistance and collision resistance) make them vital in the area of **data integrity**. For example,

1. If data is exchanged between two people (for example via email) which has been signed with a recognised hash algorithm, it will have produced a specific digest (output). If the recipient of the email runs the received message through the same algorithm and produces a different digest, they can immediately see that the message contents have been altered and should not be trusted.

2. This can be expanded to include signing certificates, such as are used for programs or code. In this instance, a software developer could produce a hash digest for a new program, creating a digital signature. A user can then download the software safely, once their operating system has generated a hash which matches that of the digital signature.

3. There may sometimes be cases where one party has some sensitive information, which cannot be revealed until a later date. In order to show that they possessed this information at the time, they can hash the information and share the hash output publicly. Based on the criteria defined on page 9, pre-image resistance says that knowing the hash output $h(x)$ should not give any clue to the value of the input x. Additionally, second pre-image resistance means that the party in possession of the information should not be able to find alternative information which would give the same hash output. Finally, collision resistance means that no other party should be able to find an alternative input which would produce the same hash output. When the time comes for the information to be revealed, the hash of the information should match what was shared initially, without ever having risked the information being leaked.

Websites also use hashing to protect **passwords**. A website obviously doesn't want to store a list of plaintext passwords for security reasons. Therefore, the passwords are hashed, and when you type in your password, its hash is checked against the stored hash digest. Password hashes are often "salted" to add additional security. Salting is where a hard-to-predict, ideally unique, random number (the "salt") is applied to the password before it is hashed [6]. This is due to the nature of passwords – they are typically short and can often be duplicated. For example, if the (unwise) password "Password123" is used and then hacked by some means, the attacker now knows the hash digest related to the input "Password123". Therefore, everyone else who had used the same password would be immediately identifiable by the same hash digest. Additionally, the attacker may be able to use this input-output pair to ascertain which hash algorithm is being used, which could be useful in breaking other common passwords. The salt adjusts the plaintext passwords pre-hashing so they will no longer be identical. So when a user enters their username and password, the salt is appended to the password and hashed again to check against the stored hash. It is therefore very important that each salt is unique, as a system-wide salt would be much less effective if even a small number of passwords were compromised.

A salt will usually be 32 or 64 bytes in size, depending on storage capacity and need for additional protection. A longer salt will increase complexity of attempted attack, and would also increase storage requirements for malicious rainbow tables.[1] However, the security of this scheme does not depend on hiding, splitting, or otherwise obscuring the salt. In practice, the salt is stored with the username and hash. So although the salt may be susceptible to discovery if a system were compromised, the fact that each password has a unique salt still means that very little has been gleaned.

How Secure are Cryptographic Hash Functions?

The security level of each algorithm is assessed relative to a birthday problem style attack, which is based on the "Birthday Problem" or "Birthday Paradox" from Probability [7]. As a reminder, the Birthday Problem asks us how many people you need in a room to find that two people share the same birthday. If we assume k people in a room, ignore leap years, and assume uniform distribution, we can say the following:

$$\Pr(\text{At Least One Shared Birthday}) = 1 - \Pr(\text{All People Having Distinct Birthdays})$$

$$\begin{aligned}\Pr(\text{All People Having Distinct Birthdays}) &= \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times ... \times \frac{365-k+1}{365} \\ &= \frac{365!}{(365-k)!365^k}\end{aligned}$$

If we replace this formula with a more generalised term, n, rather than 365 days, we get

$$1 \times \left(1 - \frac{1}{n}\right) \times \left(1 - \frac{2}{n}\right) \times \left(1 - \frac{3}{n}\right) \times \cdots \times \left(1 - \frac{(k-1)}{n}\right) \tag{2.1}$$

Looking at the Maclaurin Series (which is a simplified Taylor Series around 0) for $f(x) = e^x$, we can see that

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + ...$$

[1] Rainbow tables are pre-prepared look-up data sets whereby a large number of commonly used passwords are hashed and stored

Therefore, we can say that

$$e^x \approx 1 + x$$

We can use this to rewrite equation 2.1 above, and letting $x = -\frac{1}{n}$ this now becomes:

$$e^{\frac{0}{n}} \times e^{\frac{-1}{n}} \times e^{\frac{-2}{n}} \times ... \times e^{\frac{-(k-1)}{n}}$$

$$= e^{\frac{0-1-2-3-....-(k-1)}{n}}$$

$$= e^{\frac{0-(1+2+3+....+(k-1))}{n}}$$

We can simplify the numerator of the exponent by utilising the fact that $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ which gives us

$$e^{\frac{-(k-1)k}{2n}}$$

So now, the probability of two people sharing a birthday (or n) is:

$$1 - \text{P(All People Having Distinct Birthdays)} \leq 1 - e^{\frac{-(k-1)k}{2n}}$$

We want the left hand side to be at least 50%, so

$$0.5 \leq 1 - e^{\frac{-(k-1)k}{2n}}$$

$$\Rightarrow 0.5 \geq e^{\frac{-(k-1)k}{2n}}$$

$$\Rightarrow \ln 0.5 \geq \frac{-(k-1)k}{2n}$$

$$\Rightarrow 2n(-\ln 0.5) \leq (k-1)k$$

$(k^2 - k)$ can be considered close to k^2 for large values of k. So

$$2n(-\ln 0.5) \leq k^2$$

$$\Rightarrow k \geq \sqrt{2n(-\ln 0.5)}$$

This works out so that for k choices from n options, the probability of 2 values being is same is approximately equal to $\sqrt{n}$.

Considering this in terms of hash functions, n is now equal to the number of bits in the chosen hash. So a collision (i.e. two messages having the same hash, much like two people having the same birthday) can be found in no more than $\sqrt{2^n} = 2^{\frac{n}{2}}$ attempts. To attack pre-image resistance, it is assumed that you have to test the entire sample space and hence would require 2^n attacks. As SHA-256 outputs a 256 bit hash, the number of trials required to find a collision is $2^{\frac{256}{2}}$ or 2^{128}. For SHA 512, this increases to 2^{256} .

One might assume then that the larger the n, the more secure the hash. While this may be the case, it also means that the hash function will be much slower to run and may therefore not be desirable for practical use. Hash functions in general should be fast, for verifying connections to internet sites, downloading software etc. However slower speeds may be used for hashing passwords and other highly secure information in order to make them more difficult to break. SHA2 is still considered to be generally secure, however it is likely inevitable that it will eventually be broken, and so the next algorithm should always be prepared and ready for implementation.

Future Algorithms

There has been a move away from the traditional Merkle-Damgard construction as weaknesses continue to be exposed in algorithms which use it. In the SHA family of hashes, the next is SHA3, which follows a very different construct.

The construction behind SHA3 comes from the Keccak family of cryptographic functions, and is based on a "sponge construction" [8] [9]. BLAKE2 [10] was also a finalist in the competition for the SHA3 construct. This algorithm has a Merkle tree construction [2] which increases speed by allowing parallelism.

Whirlpool [11] is a cryptographic algorithm which does still use a Merkle-Damgard construction but combines it with the use of S-boxes. [3]
There have been no known effective attacks on any of SHA-3, BLAKE2 or Whirlpool.

Random Oracle

A random oracle [12] is a theoretical system which produces a truly random result for each input, and will give the same result each time the same input is queried. It is an idealised model which assumes the existence of a public, random function H such that all parties can obtain $H(x)$, for any desired input value x, only by interacting with an oracle computing H. There is a fun description in the Cryptography Stack Exchange [13] comparing a random oracle to a gnome in a black box. When queried, the gnome checks your input against a very large book to see if he has received this query before. If so, he provides you with the matching output, otherwise he rolls some dice to obtain a random uniform output and records the new input-output pair in the book.

Obviously this type of system cannot exist in reality, but the idea of "provable security" when assuming a "random oracle model" is often used to test workable hash functions. If we wish to prove the security of a cryptographic construction, it may be easiest to do so with the assumption of a completely random, uniform and independent result whenever the random oracle is called upon. Once the system is proven secure with this assumption, a cryptographic hash function is then inserted in place of the random oracle. This results in some mistrust of the Random Oracle Model, as there appears to be no clear definition of what makes a hash function "sufficiently good" to replicate the performance of the idealised random oracle.

2.3 Non-Cryptographic Hash Functions

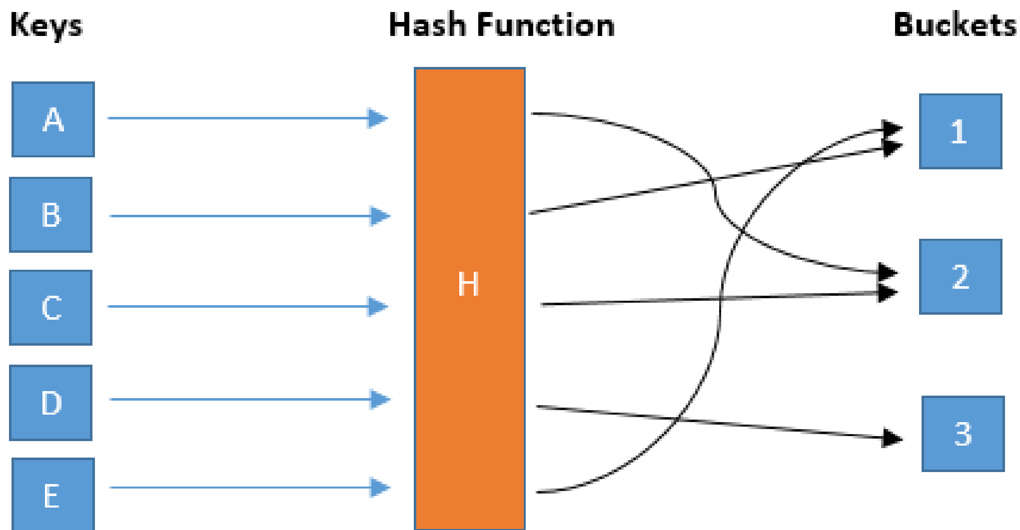
Non-cryptographic hash functions are used in many instances where security is not the main requirement, rather they focus on speed and collision resistance, usually combined with real-life input data. Hash tables, bloom filters, checksums and caches are some examples where non-cryptographic hash functions are extremely useful.

To see this in more detail, we look at the example of hash tables, whose structure is represented in Figure 2.2.

[2]The Merkle tree construction takes in all input blocks simultaneously as "leaves" and applies a hash function to each leaf. Each leaf is then combined with its neighbour and hashed again, this process continuing until the top of the "tree" is reached, giving us the final hash output

[3]S-boxes or Substitution Boxes are similar to a look-up matrix where the input is split into columns and rows to be used as a reference to find an unrelated output within the matrix.

Figure 2.2: Hash Table Structure



2.3.1 Hash Tables

The hash table structure reads in any number of input keys, runs each through the chosen hash function and then distributes the outputs among a pre-selected number of "buckets". A useful way to think of this is in terms of a library. Our library has a large number of books, let's say $1,000,000$ books. Each of these books can be defined, for example, by their name, author and year of publication. This information is now our input key. This is run through a hash function, with the output telling us to which library shelf the book has been assigned. If our library has $10,000$ shelves, each shelf should hold 100 books. When someone wishes to borrow a book, we hash the input of their chosen book, see the same output again and know exactly where to find it.

We know from the deterministic nature of hash functions that each of the keys in Figure 2.2 should produce a different digest. However, with only 3 buckets to which to map them, there needs to be a compression function to assign each digest to a bucket. This is usually done by modulo arithmetic, whereby $H(k) = N \bmod M$, where N is the bit output of the hash function and M is the number of buckets.

Traditionally, M is calculated by considering either a prime number or a power of two which is closest to the size of the hash table. The question of whether M should be a prime number or a power of two is one which has divided opinion for years [14]. Supporters of the prime number argument say that a prime number will ensure greater distribution in the output, 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. In the paper "Performance of the most common non-cryptographic hash functions" by C.Estébanez, Y.Saez, G.Recio and P. Isasi [15], a number of hash functions were tested with a variety of input types. It was noted that for data sets which contained a high number of zeros or ones, certain hash functions produced a significantly higher rate of collisions when using a power of 2 rather than a prime number to compute M. These specific functions tended to rely on

Figure 2.3: Separate Chaining

Bucket 1	6	→	21	\		
Bucket 2						
Bucket 3	33	→	8	→	18	\
Bucket 4	24	\				
Bucket 5						

multiplication or XOR operations which are vulnerable to repeating patterns and could, under certain circumstances, end up repeating values which are multiples of x. If the size of the table M has a common divisor with x, the function will continue to return multiples of this common divisor. We will come across this question again in our own tests in later chapters.

2.3.2 Collision Management

We can see from the above example of hash tables how important it is to have a uniform distribution of outputs along with strong collision resistance. If these qualities are poor, we could end up with a high number of hash outputs being mapped to few buckets, while a large number of buckets remain empty. Likewise, a good avalanche effect should ensure that similar inputs are not mapped to similar outputs. However, if the number of buckets available is of smaller size than the input key then, by the pigeon-hole principle, there must be some buckets with multiple entries, or collisions.

The two most common methods used for collision resolution are separate chaining and open addressing:

In **separate chaining**, any colliding entries are chained together into a single linked list of key-value pairs which can then be searched. Figure 2.3 shows an example of this. In this table, we have five buckets, so all outputs are assigned their bucket "mod 5". For example, Bucket 1 contains both 6 and 21 which are linked in a chain.

This is a simple method and means that we can keep adding to the hash table if the number of input keys increases. However, especially if the output distribution or collision resistance were poor, we could end up with a very long chain in one bucket, next to wasted space of unused buckets. This would practically negate the benefit of using hashing in the first place, as it reduces the process to one more closely resembling a lookup.

To add an output to a hash table, we have to "walk along" the chain of the bucket to which it hashed, and either add it to an existing slot if it matches, or add a new slot at the end. Likewise, if we wish to retrieve a result from the table, we also walk the chain until we either find the required output (successful search) or reach the end without finding it (unsuccessful search). If we assume a random output with uniform distribution, we would expect all chains to be roughly the same length of N/M where N = number of inputs and M = number of buckets. Therefore a successful search should take the average of this chain length, $\frac{N}{2M}$, whereas an unsuccessful search would require you to check the whole $\frac{N}{M}$ length. Obviously the worst hashing performance would result in all outputs hashing to one bucket, thereby creating a list of length N, and requiring O(N) work.

Open addressing can only be used in instances where the number of buckets required is unlikely to exceed the total number of items hashed. In open addressing, if the hashed-

to bucket is full, the next bucket is checked, continuing in sequence until an empty slot is found and used. When searching to retrieve an entry, the same sequence is followed until either the target record is found, or an empty slot is reached which means that there is no entry in the table to reflect that input.

The **Load Factor** is defined as the ratio of the number of objects hosted by the hash table, to the total number of available buckets (N/M). When using separate chaining, the load factor can exceed 1 without impacting much on performance. For a load factor of close to 1, you would hope that each output would hash to a different bucket and hence there would be no collisions so that, when searching for an output, there would be no chains to traverse. However, we then have to question if this is the best use of memory. Depending on the speed of your search, it may be a more efficient system overall to have a load factor > 1 and concentrate on good distribution. Your table will therefore take up much less space, and search time will hopefully be limited to $\frac{N}{M}$ at worst.

For open addressing, a load factor of 0.5 is preferred. As the load factor gets close to 1, the performance of the hash table worsens and resizing may be necessary. If the load factor reaches 1 and the table is therefore completely full, a search using open addressing would never find an empty slot, and would end up in an infinite loop.

Generally, when a hash table needs to be resized, a new hash table double the size of the existing one is created, and all original entries are rehashed and adjusted with the new modulo M.

2.3.3 The link between minimizing empty buckets and good distribution

While collisions are inevitable in a hash table with a load factor > 1, we know that we wish to minimize them. Out target is to have our outputs as evenly distributed as possible, which would mean keeping both collisions and empty buckets to a minimum. Below, we look at how minimizing the number of empty buckets is analogous to obtaining a uniform distribution.

Consider the case where X_i is a random variable which takes the value 0 if nothing is in bucket i, and 1 otherwise. Let p_i be the probability of an output landing in bucket i. We know nothing about the uniformity or otherwise of output distribution so we have no expectation for p_i.

If $n =$ the number of input keys, and $m =$number of buckets, the chance of bucket i ending up empty after n inputs is $(1 - p_i)^n$.

Therefore,

$$\mathbf{E}[X_i] = 0.(1 - p_i)^n + 1.(1 - (1 - p_i)^n)$$

i.e. we see either zero outputs in bucket i, or at least one. [4]

$$= 1 - (1 - p_i)^n$$

$$\mathbf{E}[X] = E[X_1 + X_2 + \cdots + X_m]\text{to cover all m buckets}$$

If we now search for empty buckets, and recall that $X_i = 0$ if bucket i is empty, then,

$$\mathbf{E}[\text{empty buckets}] = \mathbf{E}\Big[\sum_{i=1}^{m}(1 - X_i)\Big]$$

[4]Recall that the expectation of a discrete random variable X is given by $\mathbf{E}[\mathrm{X}] = \sum_i iPr(X = i)$

$$\mathbf{E}\Big[\sum_{i=1}^{m}(1-X_i)\Big] = m - \mathbf{E}\Big[\sum_{i=1}^{m} X_i\Big]$$
$$= m - \sum_{i=1}^{m} 1-(1-p_i)^n$$
$$= m - m + \sum_{i=1}^{m}(1-p_i)^n$$
$$= \sum_{i=1}^{m}(1-p_i)^n$$

How do we then minimize this number of empty buckets?
We use the **Lagrange Multiplier** [16].

In general terms, if we have a function $f(x, y)$ and a constraint whereby $g(x, y) = c$, the maximum and minimum values of the function f subject to the constraint g are the places where g is tangent to f. Therefore, their gradient is parallel at these points, and therefore we can say that $\nabla f(x_0, y_0) = \lambda \nabla g(x_0, y_0)$
The Lagrange Multiplier can then be written as follows

$$\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda g(x, y)$$

To find the gradient, we take the differential of L with respect to each variable and let each equal zero.

Coming back to our example above, p_i is now our only variable. So we can say

$$f(p_i) = \sum_{i=1}^{m}(1-p_i)^n$$

We know that the sum of the probabilities of all possible outcomes must total 1, i.e. $\sum_{i=1}^{m} p_i = 1$
Hence we can say that our constraint, $g(p_i) = \sum_{i=1}^{m} p_i - 1 = 0$
So in this case, the Lagrange Multiplier says

$$\mathcal{L}(p_i, \lambda) = \sum_{i=1}^{m}(1-p_i)^n - \lambda \sum_{i=1}^{m}(p_i - 1)$$

and when we differentiate this with respect to both variables we see

$$\frac{\delta \mathcal{L}}{\delta \lambda} = 1 - \sum_{i=1}^{m} p_i$$

$$\frac{\delta \mathcal{L}}{\delta p_i} = n(1-p_i)^{n-1} - \lambda \text{ , for each possible } p_i$$

Setting each of these to zero, we see first that

$$1 - \sum_{i=1}^{m} p_i = 0$$
$$\Rightarrow \sum_{i=1}^{m} p_i = 1$$

which was our original constraint.

And,

$$n(1-p_i)^{n-1} - \lambda = 0 \Rightarrow n(1-p_i)^{n-1} = \lambda$$
$$\Rightarrow (1-p_i)^{n-1} = \frac{\lambda}{n}$$
$$\Rightarrow p_i = 1 - \left(\frac{\lambda}{n}\right)^{\frac{1}{n-1}}$$

As both λ and n are constant, all p_i must be the same, which implies **uniform distribution**, from our initial aim of minimizing both collisions and empty buckets.

A similar argument could be used in order to minimise the time taken to search chains. In this case, we start from the assumption that if Y_i represents the average length of our chain, then the expected time taken to find the correct entry will be $\frac{Y_i}{2}$. We can then say that

$$\mathbf{E}[\text{Time taken to find entry in chain}]$$
$$= \sum_{i=1}^{m} \mathbf{E}[\text{Time required} \mid \text{item is in chain } i]\mathbf{Pr}(\text{item is in chain } i)$$

which is based on the probability rule which says that

$$\mathbf{E}[X] = \sum_{y} \mathbf{E}[X|Y=y]\mathbf{Pr}(Y=y)$$

As we have said that the average time taken to find the correct entry will be $\frac{Y_i}{2}$, and the probability of an entry being in bucket i is represented by p_i, then

$$\mathbf{E}[\text{Time taken to find entry in chain}] = \sum_{i=1}^{m} \mathbf{E}\left[\frac{Y_i}{2}\right].p_i$$
$$= \sum_{i=1}^{m} \frac{np_i}{2}.p_i$$
$$= \frac{n}{2}\sum_{i=1}^{m} .{p_i}^2$$

We then use the Lagrange Multiplier as above, with the same constraint that $\sum_{i=1}^{m} p_i = 1$ and we come to the conclusion that $p_i = \frac{\lambda}{n}$ which is again a constant, and again supports uniform distribution.

2.3.4 General Testing Methodology

Based on a number of publications, [15][17][18], there are four standard tests generally used to measure and compare performance of non-cryptographic hash functions:

Distribution of Output

This is measured as a comparison between the observed distribution of the algorithm and the expected distribution of a uniform output. There are a number of options which can measure this, such as Pearson's chi-squared test, the Bhattacharyya Distance (which measures the similarity of two probability distributions), or the Kullback-Leibler Divergence (which measures the divergence between two probability distributions).

Collision Resistance

A simple ratio of the number of collisions observed as a percentage of the total number of tests ran. When testing using hash tables, these collisions are defined as two outputs being mapped to the same bucket, rather than two distinct inputs hashing to the same output.

Avalanche Effect

A matrix is created for every (i, j) pair representing the effect on output bit j following a change to input bit i. The ideal is a 50% probability that each output bit j will change following a change to input bit i.

Speed

Usually measured in time taken (ms) to hash a database a specified number of times, often 1,000,000. This can be dependent on several things such as the CPU, quality of code, optimisation level of compiler etc.

2.4 Summary

Having initially introduced the basics of cryptographic hash functions in this chapter, we are now thoroughly in the world of non-cryptographic functions and hash tables. We looked briefly at collision management and the associated importance of uniform distribution, which we will return to in more detail in Section 4.1 before testing these properties in Section 5.2.

We will outline our chosen methodology for each test in more detail in Section 4.1. It would appear, at first glance, that the requirements of a uniform distribution, minimal collisions and a good avalanche effect are in fact nearly the same thing. A good avalanche effect should ensure that even very similar inputs will produce totally different outputs, which one would expect to equate to good distribution. And a good distribution should ensure that the outputs are spread evenly amongst all buckets, thereby minimising collisions!

Will a uniformly distributed algorithm ever result in very high collisions? Would there ever be a case whereby a nearly perfect avalanche effect corresponded to a poor distribution? We shall find out in Chapter 5.

Chapter 3

The FNV Hash Algorithm Family

3.1 The FNV Hash Functions

One example of a non-cryptographic hash function family is FNV, of which the current versions are FNV-1 and FNV-1a. These algorithms were created by Glenn Fowler, Landon Curt Noll and Kiem-Phong Vo. We will provide further background detail in Section 3.2. They were designed to be fast, while keeping collision rates low. They have extensive real-world uses including, according to their IETF Draft [19], DNS servers, the Twitter service, database indexing hashes, major web search / indexing engines, anti-spam filters and more. The hashes have their own website [20] which also lists further uses. This website, plus many other sources, also distinguish this algorithm from others, where sometimes only scant information is available on their creation, structure, and implementation. As an additional indicator of the widespread use of these algorithms, the FNV Primes (see detail under heading "FNV Primes" on page 22) are extensively referenced on Github, which is a collaborative code hosting platform. A search for the 32-bit FNV prime number (in decimal form) gave 30,800 code file results on Github. A search for the next prime after this number (16777633) produces just 47 results. A search for the 64-bit prime in decimal format produced 4,500 results. Hence we will use the highly popular 32-bit algorithms in our calculations.

We will start with a general overview of the FNV structure before delving into more detail in the coming pages:

FNV algorithms produce a hash of various bit sizes (32-, 64-, 128-, 256-, 512- or 1024-) depending on the availability of FNV primes.

FNV-1 has a simple structure, which could be thought of as a Merkle-Damgard structure with just one round:

- Set the initial hash value, h, to the "FNV Offset Basis"(see below section for details on Offset Bases). This acts like an Initialisation Vector as introduced in Section 2.2
- For each byte of input, b_i, first multiply h by the "FNV Prime" and then XOR it with the input byte b_i.

FNV-1a uses the same operations, but with the multiplication and XOR steps reversed. See Algorithm 1 below for the 32-bit FNV-1 and FNV-1a pseudocodes.

Multiplication for both is performed modulo 2^n where n is the bit length of the hash. The XOR is performed on the lower 8 bits of the hash.

Algorithm 1 Pseudocode for 32-bit FNV-1 and FNV-1a algorithms

```
    Prime := 0x01000193
 2: B[i] := ith input byte

 4: procedure FNV-1
        hash := 0x811c9dc5
 6:     i := zero
        for each i in B[i] do
 8:         hash := hash * Prime
            hash := hash XOR B[i]
10:         i++
        end for
12:     return hash
    end procedure
```

```
14: Prime := 0x01000193
    B[i] := ith input byte
16:
    procedure FNV-1A
18:     hash := 0x811c9dc5
        i := zero
20:     for each i in B[i] do
            hash := hash XOR B[i]
22:         hash := hash * Prime
            i++
24:     end for
        return hash
26: end procedure
```

FNV Offset Basis

The original FNV-0 algorithm had zero as its initial value, which meant that all empty messages hashed to zero. FNV-0 is therefore no longer used as a hash, but rather is now used to compute the FNV offset bases for FNV-1 and FNV-1a.

The offset bases for FNV-1 and FNV-1a are the hash (using FNV-0) of the ASCII values of the following characters:

$$\texttt{chongo} < \texttt{Landon Curt Noll} > \wedge..\wedge$$

This is the email signature of Landon Curt Noll, one of the creators of the algorithms, and acts somewhat like an initialisation vector. It can also be considered a "nothing-up-my-sleeve number" as described on page 10.

Some examples of lower order FNV offset bases in decimal / hexadecimal form are shown in Table 3.1.

Table 3.1: Offset Bases for FNV

Offset Bases		
	Decimal	Hexadecimal
32 bit	2166136261	0x811c9dc5
64 bit	14695981039346656037	0xcbf29ce484222325
128 bit	144066263297769815596495629667062367629	0x6c62272e07bb014262b821756295c58d

FNV Primes

The FNV prime is determined as follows:

Start with an integer s, $4 < s < 11$. Then let

$$n = 2^s$$

$$t = \left\lfloor \frac{(5+n)}{12} \right\rfloor \tag{3.1}$$

The n-bit FNV prime is the smallest prime number p which is of the form:

$$256^t + 2^8 + b$$

for $0 < b < 2^8$, where the number of 1 bits in b's binary representation is either 4 or 5, and where p mod $(2^{40} - 2^{24} - 1) > (2^{24} + 2^8 + 2^7)$.

The reason that s is restricted to this range is that the hash quality using a smaller s would be too low, whereas there is little need for a hash of such large size that choosing $s > 11$ would create. Additionally, primes would be more sparse at this size. This range for s creates n-bit hashes, with n ranging from 32 to 1024

The authors claim that the above criteria for FNV Primes results in better dispersion properties, while the specification for the prime mod $(2^{40} - 2^{24} - 1)$ to be greater than $(2^{24} + 2^8 + 2^7)$ is to harmonize the FNV Primes across the range of $4 < s < 11$ (see Section 3.2 for more detail).

See Table 3.2 for some lower-order FNV primes in decimal / hexadecimal form.

Table 3.2: FNV Primes

FNV Primes			
	Binary	Decimal	Hexadecimal
32 bit	$2^{24} + 2^8 + 0x93$	16777619	0x01000193
64 bit	$2^{40} + 2^8 + 0xb3$	1099511628211	0x00000100000001B3
128 bit	$2^{88} + 2^8 + 0x3b$	309485009821345068724781371	0x0000000001000000000000000000013B

If a hash-output is required in a non-standard size, *XOR folding* can be used to create it. When XOR folding, one shifts the excess high order bits down, and XORs them with the lower x-bits. In general, if a hash of k bits is required and k is not one of the sizes for which the constants are provided, one should select the smallest FNV hash of a size larger than k and then utilise XOR folding. For example, if we needed a 24-bit hash, we would calculate a 32 bit hash and XOR fold in.

3.2 Interview

With my supervisor, I interviewed Mr. Landon Curt Noll on 5th April 2022. Mr. Noll has a groundbreaking background in the area of prime numbers, having been the youngest person to find the largest known prime in 1978 at age 18. He later found a second Mersenne prime while studying number theory and was furthermore one of the "Amdahl Six" who found what was at that time the largest known prime, which was significant in that it was a rare non-Mersenne prime.

He has worked for years in the areas of security consulting and architecture, system management, cryptography, random number testing, and others. He is the founder of the International Obfuscated C Code Contest and was a member of the working group that developed the IEEE POSIX standard.

Outside of his contributions to mathematics, computer science and physics, Landon also has a keen interest in astronomy and has made astronomical observations during total solar eclipses from every continent.

Mr. Noll very kindly gave us some interesting insight into the creation of the FNV hashes.

In our original email contact with Landon, we were enquiring about the early days of FNV, as we had read on the online IETF draft that it had been based on

> "an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee by Glenn Fowler and Phong Vo back in 1991. In a subsequent ballot round: Landon Curt Noll improved on their algorithm"

We learned that the original comments to the 1991 POSIX committee were actually in relation to sum commands, so FNV initially arose out of finding a better solution to these. As further background, Landon mentioned that patents were a controversial topic at this time, following the attempt by AT&T/Bell Labs to patent SetUID. One of the aims of the authors of FNV was therefore that their algorithm should be "unpatentable", along with providing very good dispersion at high speed. Initial discussions between the three authors of FNV were notated only on restaurant paper napkins, as the details of the FNV primes were originally worked out. While Landon Curt Noll was involved in networking at the time and obviously had a strong background in primes, Phong Vo had a keen interest in linguistics. Distribution was measured mainly using real data sets, such as dictionary words (English and non-English), IP addresses, website URLs, social media content and chemistry & biology publications. In addition to speed, the authors also wanted similar strings to give widely varying results, which could be measured using the above data types. In Mr. Noll's own words, the main goal was to "fail to suck"....

We also asked Landon to expand more on the intricacies of the design of FNV, having assumed that the primes they had chosen would have bit patterns likely to give pleasing mixing properties. Landon started by explaining that XOR and multiplication were chosen as being "orthogonally different" operators. The mixing function of XOR is complemented by multiplication by a prime. A prime is preferable to a power of 2, as it will not be a multiple of word size or bit pattern and will therefore avoid repeating patterns. For the same reason, they avoided multiplication by square or cube roots: obviously if we chose our multiplier to be a square root of bit size n, two runs through the program will result in $\sqrt{n}.\sqrt{n}$ and bring us right back to multiples of bit size.

The choice of criteria for the FNV primes was to ensure better dispersion and to ensure they "propagated carries". The authors decided that the optimal prime numbers would have a small number of non-zero bits: a 1 bit roughly two-thirds of the way up, one at the eighth bit, and a small number of 1 bits in the bottom 8. The 1-bit which is roughly two-thirds of the way up should not be a multiple of the program bit-size (n).

The formula $256^t + 2^8 + b$ where $t = \lfloor\frac{(5+n)}{12}\rfloor$ is designed to ensure this distribution of bits.
For the example where b contains 4 "1-bits", this formula can be considered as follows:

$$2^{8t} + 2^8 + 2^{b_0} + 2^{b_1} + 2^{b_2} + 2^{b_3}$$

As the value of t is dependent on n, the bit-size of the FNV function, the formula for t first ensures we do not have a multiple of bit size. 2^{8t} then gives us the required 1-bit about two-thirds of the way up our prime. Multiplying by this 2^{8t} shifts the intermediate FNV hash value up by several bytes. This means that the bottom bits of the intermediate

hash values are moved up to populate the high bits, and will have a big multiplicative effect after the next byte is XOR-ed in. By multiplying by 2^8, the intermediate FNV hash value is shifted up by 1 byte. This moves the bits which were previously XOR-ed in up 1 byte, where they impact their immediate neighbour of the new octet being XOR-ed in. Multiplying by b, (where $0 < b < 2^8$ and b has 4 1-bits) means that the intermediate FNV hash value is shifted up 4 times by between 1 and 8 bits. These will then interact with the new octet being input. We can see the pattern achieved reflected in the hexadecimal expression of the FNV Primes in Table 3.2 on page 23.

The final criterion for the FNV Primes of p mod $(2^{40} - 2^{24} - 1) > (2^{24} + 2^8 + 2^7)$ was added to harmonise the choice of primes. After following the above-described criteria, there were two primes available for one of the lower order hashes. Testing showing one superior to be to the other and hence this additional requirement ensures that the correct prime is chosen in this case.

The authors had originally planned to work only on small sized functions, but demand for hash outputs of 128-bit and larger grew, and they expanded as far as 1,024-bit functions. However going any larger than this would run into issues due to prime gaps and so is not feasible. In fact, it has been found that there are no "FNV Primes" occurring between 2^{11} and 2^{20}.

Separately, there has recently been an interest in exploring non power-of-two "FNV-Style" primes, where the constraint that $n = 2^s$ is removed, and n can be any integer. For example, the FNV-Style prime generated by $n = 44$ would be $4,294,967,597$ and could potentially be used in the place of taking the 64-bit FNV hash and XOR folding.

Convolution

An interesting angle presented by Mr. Noll was the idea of convolution, where multiplication can be considered in terms of an integral. Further detail on this is provided in Appendix A.

If we consider the intermediate hash value and the FNV prime as two polynomials in base 2 then, as in a Cauchy product where the sequences are polynomials, the coefficients of the ordinary product of the polynomials are actually the convolution of the original two sequences, i.e.

$$\left(\sum_{i=0}^{\infty} a_i x^i\right) \cdot \left(\sum_{j=0}^{\infty} b_j x^j\right) = \sum_{k=0}^{\infty} c_k x^k, \text{where } c_k = \sum_{l=0}^{k} a_l b_{k-l}$$

Therefore our multiplication within the hash function can be considered a convolution and could also be represented by an integral.

Sticky State at Zero

As the FNV hashes are based primarily on multiplication and XOR functions, they are sensitive to the number zero; if the intermediate hash value were to become zero at any point and additional inputs were all zero, the hash would not change thereafter. Landon referred to this as the "sticky state" at zero. This characteristic makes it possible to find collisions if you focus on obtaining a zero step at some point in the algorithm. This is referenced in the most recent internet draft [19] which also mentions the possibility of using an unknown FNV Offset value to protect against malicious collisions.

A potential solution is to monitor the hash and take steps only if it enters this "sticky state". In this case, if the intermediate hash value at any point matched the previous value, a constant would be added to the intermediate value. Landon suggested the this constant could possibly be the FNV prime which has already been assigned in the program, as its value is already in the register and would therefore not require additional space or memory. It could be an interesting area of future study to examine the regularity with which the sticky state is an issue, versus the cost in speed required to rectify it.

The Zero Hash Challenges

The authors of FNV have an ongoing competition which asks for the shortest binary data sets found which produce a FNV hash value of zero for the various sizes of both FNV-1 and FNV-1a. It has been found, for example, that the shortest binary data set for which the 32-bit FNV-1 hash is zero is 5 bytes in length, while FNV1-a can produce a zero output with an input of just 4 bytes.

3.3 Operators

We can see from FNV's pseudocodes above that multiplication is used in conjunction with XOR. It turns out that this is a very common combination of operations in hash functions. Why are these particular operations chosen above others? In general, there are two methods to calculate a hash value from an input: calculations which **change / disrupt** the data, and calculations which **mix** the data.

We look first at which operations will effectively change the data while avoiding collisions. Consider addition and subtraction, and assume we are working mod 2^{32}. If we allow x to be our initial value at any point, and k the data being read in, is there any j, $j \neq k$ s.t. $(x+k) \bmod 2^{32} = (x+j) \bmod 2^{32}$? Considering we know that k is an input byte (and hence is 8 bits long), there cannot be a j which satisfies this. Hence we know that addition as an operation should not result in collisions. There is a similar argument for subtraction.

Multiplication and division do not perform so well. It's clear that it is possible to have more than one input which will give the same answer when multiplied by x and considered mod 2^{32}. Also, any zero input will always result in zero. Division sees similar difficulties, with the additional worry of dividing by zero. Multiplication and division are actually thought of in terms of bit shift operations. For example, if we wish to multiply an integer by 17, or $(10001)_2$, we would shift its binary form left by 5 places and then add in the integer itself again. This shift left can obviously lead to the high bits of our function being lost. If we multiply our data $\bmod 32$, any bits shifted higher than the 32nd bit will be lost. Similar is true for division, albeit shifted to the right.

Next, we consider Bitwise operations. There are three main operators here: BitwiseAnd, BitwiseOr and Bitwise XOR.

We can clearly see from Tables 3.3 and 3.4 that BitwiseAnd and BitwiseOr would both result in biased calculations, as 75% of their output is consistent. However BitwiseXOR, as seen in Table 3.5, is a good candidate for a mixing function.

In practice, BitXOR can be considered as "addition without carries" or addition mod 2. So if we require the effect of the operation to be confined to the length of the input byte, XOR will ensure no contagion outside of this. On the flip side, the effect of XOR-ing a byte into the hash function will only ever have an effect on the bottom 8 bits

Table 3.3: BitwiseAnd: Behaves like $(a \times b)$mod 2

	0	1
0	0	0
1	0	1

Table 3.4: BitwiseOr: Returns 1 when either one OR both inputs = 1

	0	1
0	0	1
1	1	1

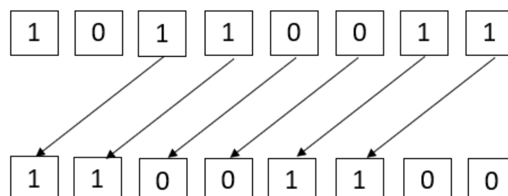
Table 3.5: BitwiseXOR: Returns 1 when inputs are different

	0	1
0	0	1
1	1	0

of the output. Hence we need to introduce a further step, to distribute this effect further up the output.

Shift and Rotate are two common mixing steps. A shift left command pushes the chosen number of bits to the left. It adds a zero at the end for each bit moved up, and any bits which "spillover" at the top are lost. While a rotate command also pushes the bits to the left, rather than losing the bits from the top, they are rotated back to the bottom. See figures 3.1 and 3.2 to compare.

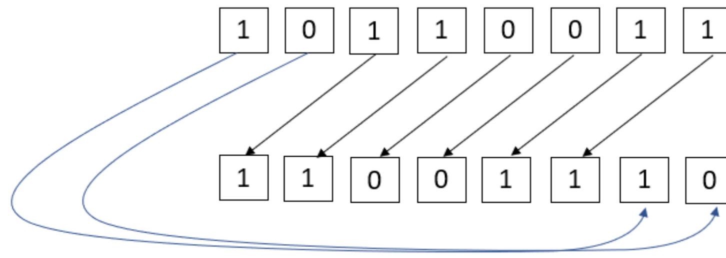
Figure 3.1: Shift Operation on 8 bits



Finally multiplication is also used as a mixing step. However the choice of multiplier is important. Multiplying by a power-of-two will just equate to a bit shift. And we know from above that, when considered mod 32, multiplication can easily lead to collisions. It is for this reason that a prime number is almost always chosen as the multiplier. However, it could be shown that any odd number could also give equal success in avoiding collisions: We wish to ensure that there is a one-to-one relationship between our input and the hash output. All invertible functions are one-to-one, and if a function is invertible then we can say that for every a, there exists a b such that $ab \equiv \mod n$. This only occurs when $\gcd(a, n) = 1$. Obviously, for any odd number a, we can say that $\gcd(a, 2^x) = 1$

While the pseudocode for the FNVs shown in Algorithm 1 above shows how they combine XOR and Multiplication, we will see further on how the Murmur2 hash (Algorithm 3 on page 32) uses a significantly higher number of mixing steps.

Figure 3.2: Rotate Operation on 8 bits



3.4 Summary

This chapter has given us a strong understanding of the FNV algorithms, from the published details on how the FNV Primes and Offsets were created, to the more nuanced background of why these particular criteria were deemed best by the authors themselves. We received some very valuable insight into the background of FNV direct from Mr. Noll, along with some interesting ideas and concepts. Finally, we also examined why certain operators are favoured by many hash functions and how they interact with each other.

In the next chapter, we shall start the process of testing and comparing these hash functions, by setting up our test framework.

Chapter 4

Design of our Test Framework

In this chapter, we will look first at what we intend to test, and then how we will do so. We wish to evaluate the performance of the FNV hash functions when used to populate hash tables. We will therefore need sets of input data to feed to the algorithms, which will then be distributed into our hash table. We will need to decide on the size of the hash table and the load factor it should have. We will also need to introduce some peer non-cryptographic hash functions in order to compare FNV's performance.

What then will we measure? We have assumed, partly based on Section 2.3.3, that the optimal distribution will be a uniform distribution. We will therefore use this as our baseline. We will also measure collision resistance by observing the number of collisions which occur, and the avalanche effect will be shown in matrix format.

4.1 Our Chosen Test Methodology

In order to rigorously compare these hash functions, we need to choose inputs which present a wide variety of potential real-world uses, an appropriate load factor and hash table size, along with a methodology for obtaining clear and reliable results.

Data Sets

We selected 4 data sets to use which are representative of a wide variety of potential inputs.

1. **Real Text: "Baby Names"**
   The top 1,000 Irish baby names for 2021 (the top 500 boys' names and top 500 girls' names), as per the Central Statistics Office [21]. Each input is just one name. As each character is read as one byte of input, this will test functions with a range from just 3 bytes (names such as Mia and Leo) up to our longest name of Christopher with 11 bytes.

2. **Real Bitstring: "IP Addresses"**
   32-bit strings composed of IP addresses taken from the log file of the personal web server maintained by my supervisor, Dr. David Malone. These are all unique IPv4 addresses which had accessed the server in chronological order during 2015.

3. **Synthetic Text: "Common Words"**
   Distinct strings of varying length each composed of 20 of the most commonly used 1,000 English words [22]

4. **Synthetic Bitstring: “Bias”**
   Strings comprising 999 `0xfe` bytes (11111110 in binary) with one `0xff` byte which 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.

These four will cover a wide range of possible inputs, from very short text to repeating patterns, which should test most aspects of our functions.

Load Factor

We chose a load factor of approximately 2. This load factor will offer us the chance to study collisions. Using our baseline assumption of uniform distribution, we would expect most buckets to contain two items each but know that this level of uniformity is unlikely. This load factor also represents a good result for a hash table, whereby one would hope that searching for an entry would reduce from O(N) lookups to O(1).

Collision Management

We will utilise separate chaining and will measure the average length of chain, plus the longest chain observed, for each bucket of our hash table.

Hash Table Size

We decided to use 1000 inputs as our base case which, with a load factor of 2, would give us 500 buckets in the hash table. However, that gives rise to the question posed above regarding the most suitable number format for a hash table; should it be a prime number, a power of two, or indeed simply the number which gives an exact load factor of 2.

To determine the relevance of this choice on our tests, we will distribute the output of each algorithm into hash tables of size (a) 500 (exact), (b) 499 (prime) and (c) 512 (power of two).

Algorithms to Test

1. We will test the FNV-1 and FNV-1a 32-bit algorithms as outlined in Chapter 3.

2. As a comparison, we will also test **DJBX33A**, one of the simplest hash functions available (see pseudocode 2 below). 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 string. In our version, the initial hash value = 0, in the way the algorithm appeared when first mooted by Chris Torek. In later versions, and as outlined by Dan Bernstein, this initial value is set to 5381 [23]. We would not expect the use of a different constant as IV to significantly alter performance and this is indeed supported by our findings.

3. The **Murmur2** 32-bit hash [24], created by Austin Appleby in 2008, is widely used and well known for its avalanche properties. The Murmur2 structure was originally created using a mixture of multiplication and rotation, hence the name. However, Austin Appleby found that multiply+rotate had a few major weaknesses when used in a hash function, so he switched to multiply+shift+xor. He thought Murmur2 was a better name than Musxmusx, so kept it.

Murmur2 starts with an initialisation vector, which is based on the length of the input XORed with a 32-bit seed number. There are two constants defined: an unsigned 32-bit number defined as m, and an integer, r. The algorithm pulls in 4 bytes of input at a time, multiplies this by m, XORs the result with a shift by r, and then multiplies by m again. The initialisation vector is multiplied by m, and then the two results are XORed together. Any leftover bytes at the end are then XORed in, before being multiplied by m.

Finally, there is one more mixing step to ensure the last few bytes are well incorporated, with XOR, shifts and multiplication. See pseudocode 3.

Murmur2 has since been improved further with Murmur3 [25] which is available in 32- or 128-bit versions and is now more commonly used.

Algorithm 2 Pseudocode for DJBX33A

```
B[i] := i^th input byte

procedure DJBX33A
    hash := 0
    i := 0
    for each i in B[i] do
        hash := hash * 33
        hash := hash + B[i]
        i++
    end for
    return hash
end procedure
```

Algorithm 3 Pseudocode for Murmur2 algorithm

```
seed := user chosen random input
data := our input
length := length of input
h := seed XOR length
m := 0x5bd1e995
r := 24

procedure MURMUR2
   while(length >=4)
   k := 4 bytes data
   k := k * m
   k := k XOR (k >> r)
   k := k * m
   h := h * m
   h := h XOR k

   data := data + 4
   length := length - 4
   EndWhile

   For remaining bytes;
   B[i] = ith byte of remaining data
   if (length%4 >= 3); h:= h XOR B[2] << 16
   if (length%4 >= 2); h:= h XOR B[1] << 8
   if (length%4 >= 1); h:= h XOR B[0]
   h := h * m

   Finalisation Step;
   h := h XOR h >> 13
   h := h * m
   h := h XOR h >> 15
   return h
end procedure
```

Distribution

We target uniform distribution to ensure an even distribution of outputs among the available buckets. Distribution will be measured 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

Uniform distribution would imply that p_i, the probability of landing in bucket i, is 1/(No. of buckets). Therefore if N = the total number of observations made, we can also say that $E_i = N.p_i$

Collisions

There are two types of collisions which could be observed. In the first instance, we have collisions whereby two inputs have resulted in identical hash outputs. We will refer to these as "pure collisions" and will examine them a little further in Section 7.3. We would expect (and hope!) for these to be quite rare and it would require a large volume of inputs to find them.

Because we are focusing on non-cryptographic hashes which are feeding a hash table, our main concern will be with collisions within the hash table, i.e. collisions mod M, where M = the size of our hash table. These collisions will be measured simply as the number of collisions observed per bucket. We will also measure the length of collision chains.

Avalanche

A good avalanche effect means that for every change to an input bit i, every output bit j should change with 50% probability. This is mooted as being important to create a more random-looking output and also to ensure that similar inputs are not mapped to similar outputs. We may question the importance of these qualities when all results are in. We should also note that this idea of a change to an input bit resulting in 50% probability of a change to an output bit could be achieved by using simple linear operators, and hence would be useless for cryptographic purposes.

Based on the work done by Bret Mulvey[2], we will build an avalanche matrix. The code we used to create the matrix entries is outlined in Algorithm 4. Each entry of this $(i \times j)$ matrix will be defined by a colour:

- Green $\Longrightarrow$ toggling input bit i resulted in a change to output bit j approximately 50% of the time.
- Red $\Longrightarrow$ the effect of toggling input bit i was a change to output bit j either 0% or 100% of the time.
- Yellow $\Longrightarrow$ a change of roughly 25% or 75%.

See Section 5.1 for precise details of how each matrix entry will be coloured.

Speed

The speed of these (and other) algorithms have been measured by many previous studies, for example [26] and [15]. We wished to focus more on the quality of the hash output, to examine collisions, study why certain patterns are seen and whether a uniform distribution is a good approximation. Hence we did not attempt to measure the speed of these functions.

Algorithm 4 Pseudocode for Avalanche Test

```
Trials:= 10000
INPUT_SIZE:= 4
OUTPUT_SIZE:= 4
hashfn:  = our chosen hash function

procedure AVALANCHE 32 x 32
   for(i = 0; i<INPUT_SIZE*8; i++)
     for(j = 0; j<OUTPUT_SIZE*8; j++)
       h[i][j] == 0

   for(Numtrials = 0; Numtrials < Trials; Numtrials++)
    input:= Character input array of size[INPUT_SIZE]
     for(i = 0; i<INPUT_SIZE; i++)
       input[i] == randomly generated byte of input data
       hash:= hashfn(input)
         for(int i = 0; i<INPUT_SIZE*8; i++)
           input[i/8] XOR (1<<i%8]
             /* This flips one bit, the flipped bit shifting by 1 place each time */
       newhash:= hashfn(input with bit flipped)
         input[i/8] XOR (1<<i%8]
             /* This flips the bit back */
       Output:= hash XOR newhash
       /* Compare the original hash output to the hash of the input with one bit flipped */
     for(j = 0; j<OUTPUT_SIZE*8; j++)
       if Output !=0 then h[i][j]++
         /* If the output bit has changed, the XOR will equal 1 and we add it to our h[i][j] count */
   for(i=0; i<INPUT_SIZE*8; i++)
     for(j=0; j<OUTPUT_SIZE*8; j++)
   printf(h[i][j])
end procedure
```

4.2 How many collisions *should* we expect?

First, we do need to bear in mind that, for hash tables, a uniform distribution is impossible to achieve unless either the load factor is less than one (and empty buckets are ignored) or else the number of buckets exactly divides the number of inputs. Likewise, collisions are inevitable once the load factor is higher than one.

As we are using separate chaining (see Figure 2.3 on page 16), any collisions will result in the formation of a chain.

For the case where we distribute outcomes amongst 500 buckets, the average fill of each bucket *should* be 2, based on 1000 trials and assuming a uniform distribution. This

can be represented as a classic "balls and bins" problem as often seen in probability theory [27].

Balls and Bins

Assume we have n balls which are thrown randomly and independently into m bins. Given that the probability that a ball lands in any one bin is $\frac{1}{m}$, then the probability of a bucket containing r balls after all n balls have been thrown can be given as follows:

$$\binom{n}{r}\left(\frac{1}{m}\right)^r\left(1-\frac{1}{m}\right)^{n-r}$$

$$=\frac{1}{r!}\frac{(n(n-1)(n-2)...(n-r+1)}{m^r}\left(1-\frac{1}{m}\right)^{n-r}$$

When n and m are large compared to r, this can be approximated to:

$$\frac{1}{r!}\left(\frac{n}{m}\right)^r e^{-n/m} \tag{4.1}$$

Considering that we expected the average fill of any bucket to be equal to $\frac{n}{m}$, we can call this our expected mean, and consider a **Poisson Distribution** [28].

A Poisson Distribution is a discrete probability distribution which 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.

The Poisson Distribution is represented as

$$\mathbf{P}(X=k)=\frac{\lambda^k e^{-\lambda}}{k!}$$

where k is the number of occurrences and λ equals the expected value of X.

This obviously has the same form as equation 4.1 above when $\lambda=\frac{n}{m}$. If we consider that our expected mean when we distribute 1,000 outputs among 500 buckets is $\lambda=2$, we can use this to estimate the probability of each bucket ending up with k inputs, where $k=0,1,2,...$

$$\mathbf{P}(X=0)=0.135$$
$$\mathbf{P}(X=1)=0.27$$
$$\mathbf{P}(X=2)=0.27$$
$$\mathbf{P}(X\geq 2)=1-\mathbf{P}(X=0)-\mathbf{P}(X=1)=0.59$$

Therefore, based on 500 buckets, we 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. We will compare these expectations to the results found later in Section 5.2.

4.3 Summary

In this chapter, we have chosen and justified our choices for our data inputs, the algorithms we wish to compare, and the techniques we will use to do so. We will use three different bucket sizes in order to judge the relevance of selecting a prime number vs a power of two, or indeed if it matters at all. We have also laid the groundwork for the results we would hope to see, assuming the Poisson Model. In the next chapter, we will see how closely the actual results reflect the expectation.

Chapter 5

Tests & Results

We can now use our chosen test methodology to examine each hash function in detail, starting with Avalanche testing.

5.1 Avalanche

Avalanche effect differs from Distribution and Collision Resistance in that it is traditionally measured independent of the underlying data. This makes sense as we do not particularly care about the actual output, rather the focus is on how the output *changes* when input bits are changed. We will therefore use a random 4-byte input when measuring this metric. Out of interest, we did also run the tests using our chosen data sets; results are shown in Appendix B.

In this section, we will measure the avalanche performance of FNV-1 and FNV-1a, with comparisons to the other algorithms. The performance will be shown in the form of a matrix, whereby

- A green square represents a change to output bit following a change to the input bit with a probability of between 45% and 55%
- An orange square represents a change to output bit with a probability in the range 25% - 45% or 55% - 75%
- A red square represents changes which occur with either less than 25% or more than 75% probability

This was measured over 10,000 trials, using randomly generated 32-bit inputs.

5.1.1 The Mechanics

To measure the avalanche effect, we will toggle one bit of our input at a time and, for each change, we will measure how many bits in our output hash changed from the original.

We will consider FNV-1a first, as it is the recommended version to use, and start with the most basic input: one byte.

In Figure 5.1, each row represents one bit of our input byte. Each column shows how the output bits (32-bit output as we are using FNV-1a 32-bit hash) changed when the input bit was toggled.

As an example, the first (top left) entry shows the when input bit i_7 was toggled, it affected output bit j_0 0% of the time, over 10,000 trials. Changing i_7 invoked a change in j_7 100% of the time, while it caused a change in j_8 50% of the time.

Figure 5.1: FNV-1a One byte Avalanche

	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}
i_7	0%	0%	0%	0%	0%	0%	0%	100%	50%	75%	37%	81%	61%	28%	86%	35%
i_6	0%	0%	0%	0%	0%	0%	100%	55%	73%	35%	81%	59%	29%	86%	44%	81%
i_5	0%	0%	0%	0%	0%	100%	54%	73%	37%	82%	60%	27%	87%	37%	81%	35%
i_4	0%	0%	0%	0%	100%	43%	78%	39%	80%	60%	26%	87%	41%	81%	40%	16%
i_3	0%	0%	0%	100%	63%	68%	34%	83%	58%	28%	86%	45%	77%	34%	16%	13%
i_2	0%	0%	100%	75%	62%	31%	85%	57%	28%	86%	43%	78%	37%	18%	10%	6%
i_1	0%	100%	49%	75%	38%	81%	60%	30%	85%	40%	79%	40%	20%	10%	5%	2%
i_0	100%	100%	50%	25%	87%	57%	28%	86%	43%	80%	41%	21%	10%	5%	2%	1%

	j_{16}	j_{17}	j_{18}	j_{19}	j_{20}	j_{21}	j_{22}	j_{23}	j_{24}	j_{25}	j_{26}	j_{27}	j_{28}	j_{29}	j_{30}	j_{31}
i_7	92%	92%	92%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
i_6	8%	8%	8%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	100%
i_5	8%	8%	8%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
i_4	8%	8%	8%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
i_3	4%	4%	4%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%
i_2	2%	2%	2%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%
i_1	2%	2%	2%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%
i_0	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%

What is actually happening?

The FNV-1a hash starts with the offset basis, then reads in the byte of input, and XORs them together. The offset basis in this case is 0x811c9dc5 and is 32 bits in length (recall Table 3.1). However, as our input byte is only 8 bits in length, it is only the 8 lowest bits of the offset basis which are affected at this stage. We then multiply by the FNV prime, which has a cluster of 1's in the lowest 8 bits, but then only has two further 1-bits in the higher 24. (Recall: FNV 32-bit Prime $= 2^{24} + 2^8 + 10010011$)

Now, we toggle one bit of the input, and XOR this new input byte with the offset basis, again affecting only the lowest 8 bits. At this stage, the only difference to the output will be the toggled bit, $2^k, 0 \leq k \leq 7$. After we then multiply by the FNV prime, the new hash result will have higher order bits which are identical to the original hash, as the bit toggle has not affected them. In fact, assuming some limited carries above the one-bit at 2^8, multiplication by the FNV prime should produce no differences to any bits in the top half at all.

Let's now consider the difference between the two hashes by subtraction (which is similar to XOR). Without loss of generality, let's say that we toggled bit 4, or 2^3. Hence the total difference between the two hashes would be $2^3 \times p$, where $p =$ FNV Prime, which could also be written as $2^{27} + 2^{11} + 2^{10} + 10011000$.

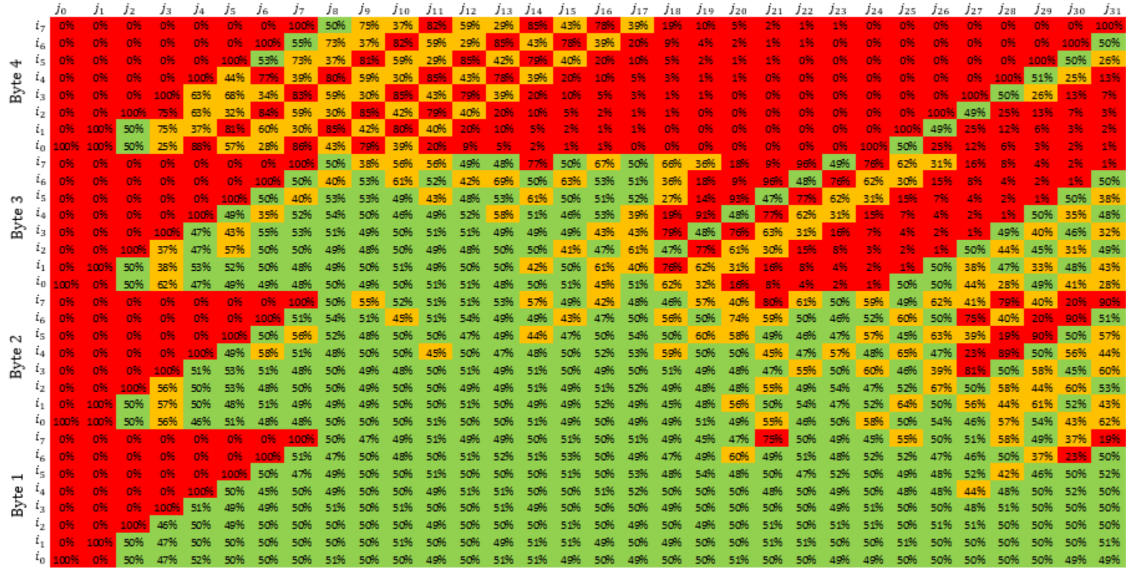
As the two hashes will always be identical for bits of lower significance than the toggled bit, their difference will always be zero, hence they change 0% of the time. The toggled input put will change the output bit 100% of the time. The section above this bit which has a mixture of 0s and 1s will show a good mix of outputs, continuing beyond 2^{11} due to carries. However there is then a sizable gap until we reach 2^{27} meaning that output changes are minimal. When we reach 2^{27}, the output will change with 100% probability, being just as bad as 0%. Likewise, the very topmost bits are never affected, meaning that the top half of the output has very poor avalanche. This can clearly be seen in Figure 5.1

FNV-1a with larger input sizes

Performance will obviously improve for larger inputs. Even by just increasing to a 2 byte input, it means that we have run the algorithm twice, and therefore the first byte to be read in will have been multiplied twice by the FNV prime. Looking at p^2, we can see that the highest bit in p, 2^{24}, will become 2^{48}, which equates to 2^{16} when we consider it mod 2^{32}. Hence we now have a flipped bit at 2^{19} (when we multiply by our toggled bit, 2^3) where there was none previously. However, as we multiply by powers of 2, we do see a lot of repeating patterns start to emerge. For example, while we saw 2^{24} squared become 2^{16}, 2^8 squared will also become 2^{16} which will then flip the bit back! So while the avalanche is improved around the middle bits, the performance remains poor on the fringes.

Next we look at the standard 32×32 matrix, by reading in 4 bytes of data.

Figure 5.2: FNV-1a 32 x 32 Avalanche



In the 4-byte example (Figure 5.2), we can see how Byte 1 at the bottom, which has now run through the algorithm four times, shows significantly more green squares than Byte 4, the last byte to be read in.

Specifically for the low order bits, the issue remains that no bit lower than the toggled bit will ever change, regardless of the number of input bytes, as the XOR between the initial hash output and the toggled hash output will always produce only zeros below the toggled bit. These zeros obviously won't be changed when multiplied by the prime. This is what gives us the distinctive "sawtooth" pattern seen in the low order bits.

Why do we read the matrix from the bottom?

You will notice that we measure avalanche from the end of the input, i.e. the first bit of the first byte read in is the final row entry. This is in order to compare inputs of varying length side-by-side. As can clearly be seen in Figure 5.2, after going through four rounds of operations, the first byte entered is now quite well mixed and producing a good avalanche matrix (apart from the lowest-order bits as discussed above), while Byte 4, which was the last byte to be introduced, is still very poor. Measuring from the end allows us to compare avalanche graphs of varying sizes (for examples of this, see Appendix B where we used our various data sets as inputs). Figure 5.3 shows how performance improves from the last byte in red, be it Byte 6 or Byte 3, in equal measure per byte.

Figure 5.3: Measuring matrices of varying length from the end

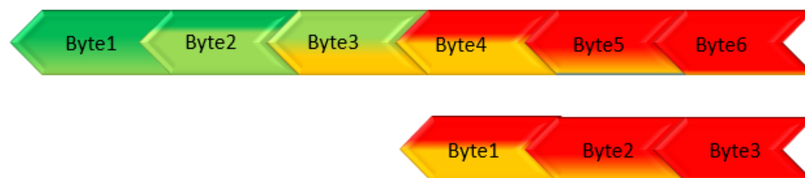
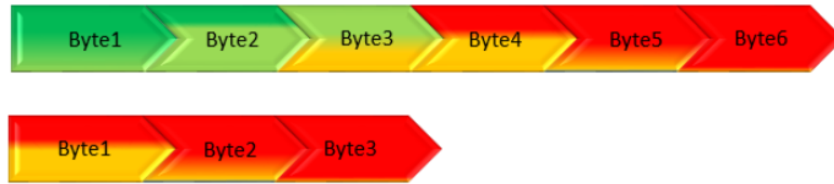


Figure 5.4 shows how, if we measured from Byte 1, the comparison between outputs of different length would not be so clear.

Figure 5.4: Measuring from Byte 1



Comparison to FNV-1

The performance of the FNV-1 hash is significantly worse in the special case of a one-byte input. As the order of operations is the reverse of FNV-1a, the first step in this hash is to multiply the offset basis by the prime. This is then XORed with the input byte. Obviously then, when we toggle a bit in the input byte, the result from the first step (offset $\times$ prime) remains identical, whereas the second step will differ only by that toggled bit. Hence we see a sea of zeros apart from one 1-bit we toggled which changes with 100% probability, giving us an entirely red 8×32 matrix.

The pattern of the two byte input for FNV-1 closely resembles that of the one-byte input for FNV-1a, as it's only by the time we reach the second input byte that $p \times 2^k$ is the difference. In fact, the entire matrix for FNV-1 appears "one byte behind" that of FNV-1a.

Figure 5.5: FNV-1 32 x 32 Avalanche

		j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	j_{17}	j_{18}	j_{19}	j_{20}	j_{21}	j_{22}	j_{23}	j_{24}	j_{25}	j_{26}	j_{27}	j_{28}	j_{29}	j_{30}	j_{31}
Byte 4	i_7	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_6	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_5	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_4	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_3	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_2	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_1	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	i_0	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3	i_7	0%	0%	0%	0%	0%	0%	0%	100%	52%	75%	37%	81%	59%	30%	85%	43%	79%	39%	20%	10%	5%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	100%
	i_6	0%	0%	0%	0%	0%	0%	100%	55%	73%	36%	82%	59%	29%	85%	42%	79%	39%	19%	10%	5%	2%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	49%
	i_5	0%	0%	0%	0%	0%	100%	53%	73%	37%	82%	59%	30%	85%	43%	78%	39%	20%	9%	5%	2%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	50%	26%
	i_4	0%	0%	0%	0%	100%	44%	78%	40%	80%	60%	31%	84%	42%	79%	39%	20%	10%	5%	2%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	50%	26%	13%
	i_3	0%	0%	0%	100%	62%	69%	34%	83%	59%	29%	85%	43%	78%	40%	20%	10%	5%	2%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	50%	25%	13%	6%
	i_2	0%	0%	100%	74%	62%	31%	85%	57%	29%	86%	43%	78%	40%	20%	10%	5%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	50%	25%	13%	7%	3%
	i_1	0%	100%	50%	75%	38%	81%	61%	31%	85%	42%	79%	39%	20%	10%	5%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	49%	25%	13%	6%	3%	2%
	i_0	100%	100%	52%	26%	87%	57%	29%	85%	43%	78%	39%	19%	10%	5%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	100%	50%	25%	12%	6%	3%	2%	1%
Byte 2	i_7	0%	0%	0%	0%	0%	0%	0%	100%	50%	38%	56%	57%	50%	48%	77%	50%	68%	51%	66%	35%	18%	9%	96%	48%	77%	61%	31%	15%	8%	4%	2%	1%
	i_6	0%	0%	0%	0%	0%	0%	100%	50%	39%	54%	61%	51%	42%	69%	50%	64%	51%	52%	36%	18%	9%	95%	48%	76%	63%	32%	15%	8%	4%	2%	1%	50%
	i_5	0%	0%	0%	0%	0%	100%	50%	39%	54%	54%	50%	42%	48%	53%	62%	50%	52%	52%	28%	14%	93%	48%	76%	62%	31%	15%	8%	4%	2%	1%	50%	38%
	i_4	0%	0%	0%	0%	100%	49%	34%	52%	55%	50%	45%	49%	52%	58%	51%	47%	53%	37%	19%	90%	48%	75%	62%	31%	15%	8%	4%	2%	1%	50%	36%	47%
	i_3	0%	0%	0%	100%	47%	43%	55%	52%	49%	48%	51%	52%	51%	50%	49%	49%	42%	43%	78%	47%	76%	62%	31%	16%	8%	4%	2%	1%	50%	41%	46%	32%
	i_2	0%	0%	100%	38%	47%	58%	51%	50%	49%	49%	51%	49%	49%	50%	49%	40%	47%	62%	47%	76%	62%	32%	16%	8%	4%	2%	1%	49%	44%	45%	32%	48%
	i_1	0%	100%	50%	37%	53%	53%	51%	49%	50%	51%	50%	49%	50%	49%	42%	52%	59%	40%	75%	62%	31%	16%	8%	4%	2%	1%	50%	38%	47%	34%	48%	42%
	i_0	100%	0%	49%	62%	47%	49%	49%	48%	51%	50%	49%	51%	49%	46%	52%	52%	44%	51%	61%	31%	15%	8%	4%	2%	1%	50%	50%	44%	29%	49%	41%	28%
Byte 1	i_7	0%	0%	0%	0%	0%	0%	0%	100%	50%	56%	52%	50%	52%	52%	57%	50%	43%	49%	48%	55%	40%	80%	60%	49%	59%	51%	61%	41%	79%	40%	20%	90%
	i_6	0%	0%	0%	0%	0%	0%	100%	49%	54%	51%	45%	51%	54%	49%	49%	44%	47%	50%	57%	50%	74%	60%	49%	45%	52%	59%	50%	75%	40%	21%	89%	50%
	i_5	0%	0%	0%	0%	0%	100%	50%	56%	52%	46%	51%	49%	46%	48%	44%	48%	50%	54%	49%	60%	58%	50%	46%	48%	58%	45%	63%	40%	21%	90%	50%	58%
	i_4	0%	0%	0%	0%	100%	50%	59%	50%	47%	51%	50%	45%	49%	45%	48%	50%	51%	53%	59%	49%	50%	45%	47%	57%	50%	63%	48%	24%	88%	49%	57%	43%
	i_3	0%	0%	0%	100%	51%	53%	51%	48%	50%	50%	48%	49%	52%	49%	50%	51%	50%	51%	51%	51%	49%	48%	56%	47%	61%	47%	39%	80%	49%	58%	45%	60%
	i_2	0%	0%	100%	56%	50%	53%	47%	49%	48%	48%	50%	50%	50%	49%	50%	48%	50%	51%	50%	48%	47%	55%	50%	53%	46%	52%	66%	50%	59%	45%	59%	54%
	i_1	0%	100%	50%	57%	51%	48%	50%	49%	49%	50%	50%	50%	51%	50%	49%	50%	52%	49%	46%	48%	56%	49%	54%	45%	51%	63%	50%	57%	44%	61%	52%	43%
	i_0	100%	100%	50%	56%	47%	50%	49%	48%	50%	50%	51%	51%	50%	50%	50%	51%	50%	48%	48%	51%	50%	54%	46%	50%	58%	50%	54%	45%	58%	54%	42%	63%

Comparison to DJBX33A

Why are these functions exhibiting poor avalanche behaviour, especially in the case where multiplication is the first operation done? To compare, we looked at another hash function

which relies heavily on multiplication: DJBX33A.
Similar to FNV-1, the first step in this hash is multiplication. However while FNV-1 starts from an initialisation vector multiplied by a prime, DJBX33A starts from zero and multiplies by 33. Hence the first run of the algorithm actually just adds the input byte to zero. So in our one-byte example (see Figure 5.6), the hash actually produces just the input byte itself. So when we toggle one bit in the input, the resulting hash will reflect the same, and the difference between the two will just be this toggled bit and zero everywhere else.

Figure 5.6: DJBX33A One byte Avalanche

	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	j_{17}	j_{18}	j_{19}	j_{20}	j_{21}	j_{22}	j_{23}	j_{24}	j_{25}	j_{26}	j_{27}	j_{28}	j_{29}	j_{30}	j_{31}
i_7	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_6	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_5	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_4	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_3	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_2	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_1	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
i_0	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Likewise, regardless of the number of bytes of input, the difference between the original hash and the toggled hash will always be zeros below the toggled bit, giving the same jagged pattern as FNV in the lowest level bits. In relation to higher level bits, its performance is significantly worse than both FNVs, as it multiplies by 33 (or 00100001) rather than a 32-bit prime. Hence, even with some carries, the impact of multiplication will take a very long time to reach any bit in the top half.
Where this hash differs from the FNVs however is that when the input size increases, the avalanche improves slightly for all bytes, including the final one to be read in. Why does it differ in this way? This is due to the use of addition rather than XOR in conjunction with multiplication. While XOR will only either flip the bit or leave it alone, addition can result in a carry, which gives the distinctive red-green-orange pattern seen in Figure 5.7. As with FNV, a toggled input bit i_k will toggle the corresponding output bit j_k with 100% probability. However, due to the addition carries, bit $j_{(k+1)}$ will switch with 50% probability, and bit $j_{(k+2)}$ will switch with roughly 25% probability and so on.

Note: To be confident in proceeding with the version of DJBX33A which utilises zero as the initial value, we examined how the avalanche behaviour would change if we used IV = 5381 as suggested by D. Bernstein instead (see description in Subsection 4.1 on page 30. As we can see in Figure 5.8, the difference is minimal, so we use IV = 0 for all future calculations.

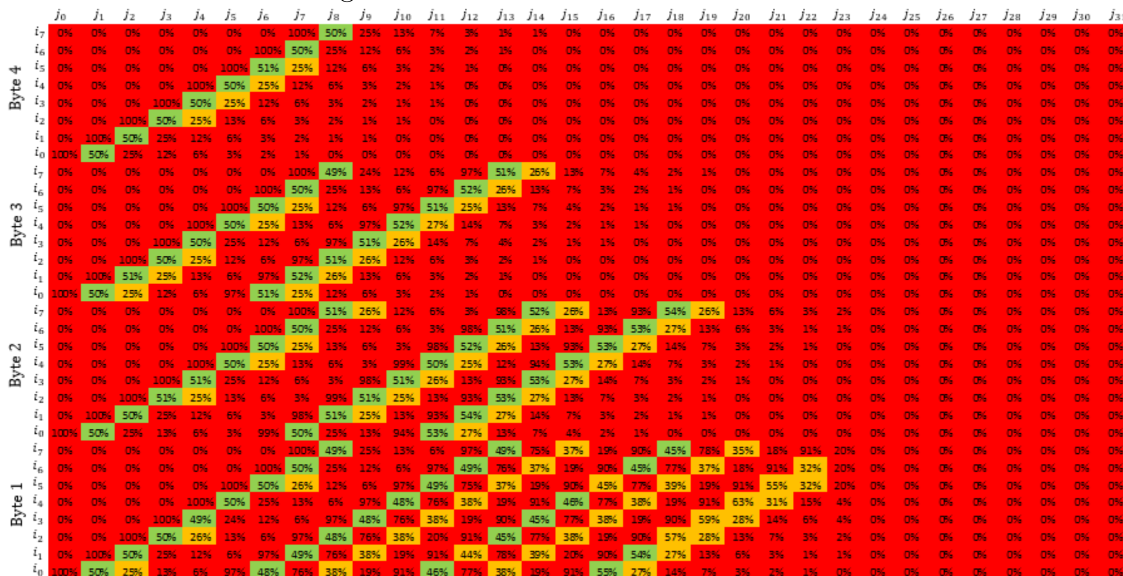
Comparison to Murmur2

For contrast, we then ran the Murmur2 hash.

As seen in the pseudocode on page 32, Murmur2 involves a much more thorough mixing step than any of the other algorithms. This creates a highly randomised looking function which translates to a fairly perfect avalanche effect as seen in Figure 5.9.

We should note that Murmur2 differs from our other functions in that it pulls in four bytes of data at a time. For inputs of less than 4 bytes, these mixing steps are bypassed and essentially only the finalisation step is applied to the input. However, because even the finalisation step involves XOR, shift and multiplication, the one-byte input still produces an relatively acceptable avalanche performance — see Figure 5.10.

Figure 5.7: DJBX33A 32 x 32 Avalanche



A quick look at FNV-1a 64-bit hash

As a final point of interest in this section, we examined the pattern displayed by using FNV-1a 64-bit hash function. While we will only use 32-bit hashes in our testing, we wished to check if the results seen so far would extrapolate to 64-bit hashes. As you can see in Figure 5.11, the same patterns do repeat, almost looking just like the 32-bit algorithm (Figure 5.2) run twice.

Figure 5.8: DJBX33A 32 x 32 Avalanche with IV = 5381

	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	j_{17}	j_{18}	j_{19}	j_{20}	j_{21}	j_{22}	j_{23}	j_{24}	j_{25}	j_{26}	j_{27}	j_{28}	j_{29}	j_{30}	j_{31}
Byte 4 i_7	0%	0%	0%	0%	0%	0%	0%	100%	50%	25%	12%	6%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_6	0%	0%	0%	0%	0%	0%	100%	51%	26%	13%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_5	0%	0%	0%	0%	0%	100%	50%	26%	13%	6%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_4	0%	0%	0%	0%	100%	49%	25%	12%	6%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_3	0%	0%	0%	100%	50%	25%	13%	6%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_2	0%	0%	100%	51%	25%	12%	6%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_1	0%	100%	50%	26%	13%	6%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 4 i_0	100%	50%	25%	13%	6%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_7	0%	0%	0%	0%	0%	0%	0%	100%	51%	26%	13%	7%	97%	52%	27%	14%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_6	0%	0%	0%	0%	0%	0%	100%	50%	25%	13%	7%	97%	51%	26%	13%	6%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_5	0%	0%	0%	0%	0%	100%	51%	25%	13%	6%	97%	51%	26%	13%	6%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_4	0%	0%	0%	0%	100%	51%	25%	13%	7%	97%	52%	27%	13%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_3	0%	0%	0%	100%	50%	25%	13%	6%	97%	52%	26%	13%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_2	0%	0%	100%	50%	25%	13%	7%	97%	52%	26%	14%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_1	0%	100%	51%	26%	13%	7%	97%	52%	26%	13%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 3 i_0	100%	50%	25%	13%	6%	97%	51%	26%	13%	6%	3%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_7	0%	0%	0%	0%	0%	0%	0%	100%	49%	25%	13%	6%	3%	98%	51%	26%	13%	94%	53%	27%	14%	8%	3%	2%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_6	0%	0%	0%	0%	0%	0%	100%	50%	25%	12%	7%	3%	98%	50%	26%	13%	93%	54%	27%	14%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_5	0%	0%	0%	0%	0%	100%	49%	25%	12%	6%	3%	99%	51%	26%	13%	94%	53%	26%	13%	6%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_4	0%	0%	0%	0%	100%	51%	25%	13%	6%	3%	98%	50%	25%	12%	94%	53%	27%	13%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_3	0%	0%	0%	100%	50%	25%	13%	6%	3%	99%	51%	26%	13%	93%	53%	26%	14%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_2	0%	0%	100%	49%	24%	12%	6%	3%	99%	51%	26%	13%	93%	53%	27%	14%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_1	0%	100%	50%	26%	13%	7%	3%	98%	51%	25%	13%	93%	53%	27%	13%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 2 i_0	100%	50%	25%	13%	6%	3%	98%	50%	25%	13%	93%	53%	27%	13%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_7	0%	0%	0%	0%	0%	0%	0%	100%	50%	25%	12%	6%	97%	48%	76%	38%	19%	90%	45%	77%	38%	20%	90%	48%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_6	0%	0%	0%	0%	0%	0%	100%	50%	25%	13%	6%	97%	48%	76%	37%	19%	90%	44%	78%	39%	19%	89%	85%	48%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_5	0%	0%	0%	0%	0%	100%	51%	26%	13%	7%	97%	49%	76%	38%	19%	90%	45%	78%	39%	20%	90%	40%	15%	3%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_4	0%	0%	0%	0%	100%	50%	24%	12%	6%	97%	48%	76%	38%	19%	90%	46%	76%	38%	19%	91%	58%	31%	14%	3%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_3	0%	0%	0%	100%	51%	25%	13%	6%	97%	48%	76%	38%	18%	91%	45%	77%	38%	19%	91%	51%	26%	14%	4%	3%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_2	0%	0%	100%	50%	25%	12%	6%	97%	49%	75%	38%	19%	91%	46%	77%	39%	19%	90%	55%	28%	14%	8%	3%	2%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_1	0%	100%	50%	26%	13%	6%	97%	48%	76%	38%	19%	90%	44%	78%	39%	20%	90%	56%	28%	14%	7%	4%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%
Byte 1 i_0	100%	50%	25%	13%	6%	97%	48%	76%	37%	18%	91%	45%	78%	39%	20%	90%	55%	28%	13%	7%	3%	2%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%

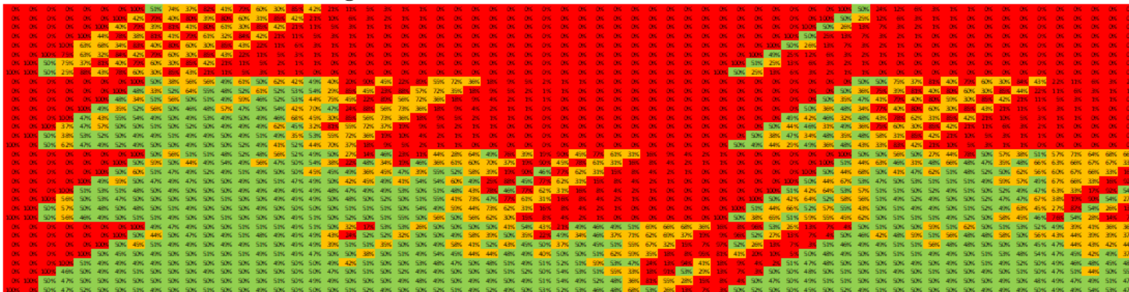
Figure 5.9: Murmur2 32 x 32 Avalanche

	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	j_{17}	j_{18}	j_{19}	j_{20}	j_{21}	j_{22}	j_{23}	j_{24}	j_{25}	j_{26}	j_{27}	j_{28}	j_{29}	j_{30}	j_{31}
Byte 4 i_7	49%	51%	50%	50%	51%	49%	50%	49%	50%	50%	50%	50%	51%	50%	50%	51%	49%	50%	50%	50%	50%	51%	50%	49%	50%	50%	50%	49%	50%	50%	51%	50%
Byte 4 i_6	50%	51%	50%	50%	50%	50%	49%	50%	50%	51%	50%	49%	50%	50%	50%	49%	50%	50%	50%	50%	50%	51%	50%	50%	50%	49%	51%	50%	50%	50%	51%	50%
Byte 4 i_5	50%	50%	50%	49%	50%	50%	49%	50%	50%	50%	50%	49%	50%	51%	51%	50%	50%	52%	50%	50%	51%	50%	50%	50%	50%	50%	50%	51%	50%	50%	51%	49%
Byte 4 i_4	50%	50%	50%	49%	50%	50%	50%	50%	51%	50%	51%	50%	50%	51%	50%	50%	50%	50%	51%	50%	50%	51%	49%	51%	50%	51%	49%	51%	50%	50%	50%	50%
Byte 4 i_3	50%	50%	50%	49%	49%	51%	50%	50%	50%	50%	50%	50%	51%	50%	50%	51%	50%	50%	49%	50%	50%	50%	50%	50%	50%	50%	50%	50%	49%	50%	51%	50%
Byte 4 i_2	50%	50%	50%	50%	49%	51%	50%	50%	49%	50%	50%	51%	50%	51%	51%	50%	49%	50%	50%	50%	50%	50%	50%	51%	51%	50%	50%	49%	50%	51%	50%	50%
Byte 4 i_1	50%	50%	49%	50%	49%	50%	49%	51%	51%	50%	51%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	51%	51%	51%	49%	50%	50%	50%	50%	50%	49%
Byte 4 i_0	50%	50%	50%	50%	51%	50%	51%	49%	51%	50%	50%	50%	49%	50%	51%	50%	50%	50%	51%	51%	50%	51%	51%	49%	50%	50%	50%	50%	51%	51%	50%	50%
Byte 3 i_7	50%	50%	50%	50%	49%	50%	49%	50%	50%	50%	50%	50%	50%	49%	51%	51%	50%	50%	50%	49%	51%	50%	50%	49%	50%	50%	50%	50%	50%	51%	51%	50%
Byte 3 i_6	49%	50%	51%	49%	50%	50%	50%	50%	49%	50%	50%	50%	50%	50%	51%	50%	49%	50%	50%	49%	50%	50%	51%	50%	51%	49%	50%	50%	49%	51%	50%	50%
Byte 3 i_5	50%	49%	50%	50%	50%	49%	50%	51%	51%	49%	51%	50%	49%	50%	50%	51%	51%	50%	50%	50%	49%	50%	49%	50%	51%	49%	51%	50%	51%	51%	50%	50%
Byte 3 i_4	50%	50%	50%	50%	51%	50%	50%	50%	51%	49%	51%	49%	49%	50%	49%	51%	49%	51%	50%	50%	50%	50%	51%	50%	49%	50%	50%	49%	50%	50%	49%	49%
Byte 3 i_3	49%	50%	50%	50%	49%	50%	50%	50%	50%	50%	49%	50%	49%	50%	50%	50%	50%	49%	50%	50%	50%	50%	50%	50%	50%	50%	49%	50%	50%	49%	51%	49%
Byte 3 i_2	50%	50%	49%	50%	49%	51%	50%	50%	49%	50%	50%	49%	50%	50%	50%	50%	50%	50%	49%	50%	49%	50%	51%	49%	50%	50%	51%	51%	50%	50%	50%	50%
Byte 3 i_1	50%	50%	51%	50%	50%	50%	50%	50%	50%	51%	50%	50%	49%	51%	50%	50%	49%	50%	49%	51%	50%	50%	50%	50%	50%	50%	50%	50%	49%	50%	49%	50%
Byte 3 i_0	51%	50%	51%	50%	50%	51%	50%	50%	50%	50%	49%	50%	50%	50%	50%	50%	50%	50%	49%	50%	49%	50%	50%	51%	50%	50%	50%	50%	50%	50%	50%	50%
Byte 2 i_7	50%	50%	50%	51%	51%	50%	49%	50%	50%	50%	49%	50%	50%	50%	51%	50%	50%	49%	50%	50%	49%	49%	51%	50%	50%	50%	50%	50%	49%	50%	49%	50%
Byte 2 i_6	49%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	51%	50%	50%	50%	50%	49%	50%	50%	50%	50%	50%	50%	49%	50%	50%	50%	50%	50%
Byte 2 i_5	50%	48%	49%	50%	49%	50%	50%	51%	50%	50%	50%	50%	50%	50%	49%	51%	49%	49%	52%	51%	49%	51%	50%	50%	50%	50%	50%	51%	50%	50%	50%	50%
Byte 2 i_4	49%	50%	50%	49%	50%	51%	50%	50%	49%	51%	50%	50%	50%	50%	50%	50%	50%	50%	50%	49%	50%	50%	50%	51%	50%	50%	50%	50%	50%	50%	50%	50%
Byte 2 i_3	49%	50%	51%	50%	50%	50%	50%	50%	50%	50%	50%	50%	51%	50%	50%	50%	50%	50%	50%	50%	50%	50%	51%	50%	50%	50%	50%	51%	51%	50%	50%	50%
Byte 2 i_2	50%	50%	51%	50%	50%	50%	50%	50%	50%	49%	50%	50%	50%	50%	51%	50%	50%	51%	50%	49%	51%	51%	50%	50%	50%	50%	50%	50%	50%	49%	50%	51%
Byte 2 i_1	50%	50%	50%	50%	49%	50%	50%	49%	50%	49%	50%	51%	50%	50%	50%	50%	50%	49%	50%	50%	50%	50%	49%	50%	49%	49%	50%	49%	51%	51%	49%	51%
Byte 2 i_0	50%	50%	49%	50%	50%	49%	50%	50%	50%	50%	49%	51%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%	51%	50%	51%	50%	50%	50%	51%	50%	50%	51%
Byte 1 i_7	50%	50%	50%	50%	48%	50%	50%	51%	51%	50%	50%	49%	50%	50%	50%	49%	50%	49%	51%	50%	49%	50%	50%	49%	51%	50%	50%	51%	51%	50%	50%	50%
Byte 1 i_6	50%	51%	50%	50%	50%	50%	51%	50%	49%	50%	49%	50%	49%	51%	49%	51%	50%	50%	50%	50%	50%	50%	51%	50%	50%	50%	50%	49%	50%	50%	51%	50%
Byte 1 i_5	49%	50%	49%	50%	51%	50%	49%	50%	50%	50%	50%	51%	50%	50%	50%	50%	51%	50%	50%	50%	50%	51%	51%	50%	50%	51%	49%	50%	50%	50%	50%	50%
Byte 1 i_4	50%	50%	50%	49%	50%	49%	50%	50%	50%	50%	50%	50%	49%	51%	50%	50%	50%	50%	50%	50%	50%	49%	50%	50%	50%	50%	50%	50%	51%	50%	50%	50%
Byte 1 i_3	50%	50%	49%	51%	50%	49%	50%	50%	49%	50%	50%	51%	50%	50%	50%	51%	49%	51%	51%	51%	51%	50%	50%	51%	51%	50%	50%	49%	49%	50%	49%	50%
Byte 1 i_2	50%	50%	51%	50%	49%	51%	50%	50%	51%	51%	50%	50%	50%	50%	50%	51%	50%	51%	49%	49%	50%	50%	50%	50%	50%	50%	49%	50%	51%	50%	50%	51%
Byte 1 i_1	51%	50%	50%	50%	50%	50%	50%	50%	49%	50%	49%	50%	50%	50%	50%	50%	51%	50%	51%	51%	51%	49%	50%	51%	51%	50%	49%	50%	50%	50%	50%	50%
Byte 1 i_0	50%	50%	50%	50%	50%	50%	49%	49%	51%	50%	51%	49%	50%	50%	50%	50%	50%	50%	50%	51%	50%	50%	49%	50%	50%	50%	50%	50%	49%	51%	50%	50%

Figure 5.10: Murmur2 One byte Avalanche

	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	j_{17}	j_{18}	j_{19}	j_{20}	j_{21}	j_{22}	j_{23}	j_{24}	j_{25}	j_{26}	j_{27}	j_{28}	j_{29}	j_{30}	j_{31}
i_7	48%	50%	59%	47%	47%	46%	53%	59%	53%	54%	58%	53%	54%	59%	52%	50%	55%	43%	51%	53%	45%	48%	53%	43%	47%	49%	52%	50%	58%	47%	53%	48%
i_6	55%	48%	38%	51%	51%	45%	53%	51%	45%	44%	46%	48%	50%	52%	56%	50%	48%	52%	44%	48%	47%	42%	52%	53%	45%	52%	51%	51%	44%	50%	44%	57%
i_5	41%	46%	48%	55%	47%	55%	56%	48%	56%	53%	53%	47%	53%	51%	46%	49%	51%	54%	50%	49%	51%	53%	53%	51%	53%	52%	48%	53%	45%	56%	48%	49%
i_4	59%	50%	44%	46%	54%	52%	43%	46%	45%	43%	46%	58%	49%	44%	45%	46%	51%	48%	46%	55%	56%	49%	46%	54%	48%	53%	51%	56%	49%	50%	47%	50%
i_3	54%	56%	51%	44%	51%	49%	50%	49%	47%	42%	45%	52%	53%	49%	49%	44%	42%	47%	60%	53%	56%	56%	50%	49%	55%	50%	44%	52%	54%	48%	48%	49%
i_2	45%	53%	47%	51%	55%	48%	52%	57%	55%	56%	51%	46%	49%	49%	61%	51%	59%	52%	50%	50%	52%	53%	42%	57%	50%	58%	44%	47%	53%	51%	53%	48%
i_1	49%	49%	49%	52%	49%	53%	48%	43%	47%	43%	56%	43%	51%	46%	46%	50%	49%	46%	56%	49%	56%	52%	48%	48%	53%	42%	46%	43%	54%	48%	54%	52%
i_0	52%	54%	44%	55%	56%	55%	46%	51%	53%	52%	52%	55%	45%	43%	50%	57%	52%	42%	49%	49%	45%	49%	53%	51%	49%	48%	48%	50%	53%	45%	50%	46%

Figure 5.11: FNV-1a 32 x 64 Avalanche



5.2 Distribution and Collisions

To run these tests, we created a program which took each of our chosen data sets as input, hashed the data using each of the four algorithms described above, and calculated the result mod M, where M is the number of buckets. We tested for various choices of M: $M = 500$ (load factor of exactly 2), $M = 499$ (a prime number close to 500) or $M = 512$ (a power of two close to 500).

For each data set, we noted how many outputs ended up in each bucket and we counted the number of collisions (i.e. buckets with more than one item), the number of empty buckets and also the length of the chains in buckets with two or more entries.

Distribution is assessed by the chi-squared p-value, which can be thought of as the probability of obtaining a chi-square value as large as was found but with the data still supporting the hypothesis, i.e. the probability of obtaining these results assuming the null hypothesis (uniform distribution) is true.

The following tables show the results of these tests. Each input data set has a separate table. Taking the *Baby Names* data set (Table 5.1) as an example, we can see that the results are shown in three "panels", the first using 500 buckets, then 499 buckets and finally 512 buckets. For each of these bucket sizes, we show first the number of collisions that were observed. We then measure the chain length where collisions occurred, showing both the average length of chain, and the longest chain observed. Next we counted the number of buckets which received no outputs. Finally, to measure uniformity of distribution, we show the chi-squared p-value. This value is a probability measure, so it will read between zero and one. A high probability is supportive of the null hypothesis, which is uniform distribution. For example, if we look at the results when the *Baby Names* data set was hashed using FNV-1 and distributed among 500 buckets (highlighted in yellow), we see that there were 69 empty buckets, alongside 297 collisions. Recall that the Poisson Distribution (Subsection 4.2) suggested we should expect to see 68 empty buckets and 297 collisions, so our results for this particular test are very accurate. The longest chain was 7 outputs long, while the average chain length was 2.92. The chi-squared p-value of 55% is supportive of the null hypothesis.

Any notable outlying results are highlighted in red across all tables.

Table 5.1: Top 1000 Baby Names in Ireland, 2021: Distribution and Collision Resistance

500 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	309	297	277	290
Average Chain Length	2.86	2.92	3.04	2.96
Longest Chain	7	7	7	9
Empty Buckets	75	69	64	68
Chi-squared p-value	0.37	0.55	0.08	0.07
499 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	292	304	291	301
Average Chain Length	2.97	2.87	2.93	2.91
Longest Chain	8	7	9	7
Empty Buckets	74	66	62	73
Chi-squared p-value	0.04	0.75	0.52	0.45
512 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	297	304	294	294
Average Chain Length	2.83	2.85	2.94	2.94
Longest Chain	6	8	7	7
Empty Buckets	56	74	83	81
Chi-squared p-value	0.99	0.61	0.17	0.16

Table 5.2: Permutations of the 1000 Most Common English Words: Distribution and Collision Resistance

500 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	301	298	298	293
Average Chain Length	2.85	2.90	2.91	2.94
Longest Chain	6	6	6	9
Empty Buckets	57	66	69	69
Chi-squared p-value	0.94	0.73	0.47	0.04
499 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	300	304	294	302
Average Chain Length	2.91	2.86	2.92	2.88
Longest Chain	8	8	8	7
Empty Buckets	73	63	63	67
Chi-squared p-value	0.20	0.79	0.35	0.70
512 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	299	292	290	291
Average Chain Length	2.87	2.91	2.93	2.93
Longest Chain	6	7	7	7
Empty Buckets	71	70	73	74
Chi-squared p-value	0.94	0.65	0.14	0.46

Table 5.3: Synthetic Bit String Biased (87.5%) Towards 1-bits: Distribution and Collision Resistance

500 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	232	223	286	125
Average Chain Length	4.25	4.37	2.98	8.00
Longest Chain	10	11	7	17
Empty Buckets	254	251	65	375
Chi-squared p-value	0	0	0.25	0
499 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	293	298	293	292
Average Chain Length	2.97	2.91	2.97	2.94
Longest Chain	7	6	10	8
Empty Buckets	76	68	76	66
Chi-squared p-value	0.06	0.64	0.02	0.25
512 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	256	256	307	16
Average Chain Length	3.91	3.91	2.82	62.50
Longest Chain	4	4	8	63
Empty Buckets	256	256	71	496
Chi-squared p-value	0	0	0.58	0

Table 5.4: IP addresses: Distribution and Collision Resistance

500 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	315	310	291	308
Average Chain Length	2.76	2.79	2.99	2.78
Longest Chain	6	8	9	6
Empty Buckets	53	54	79	49
Chi-squared p-value	1.00	0.99	0.01	1.00
499 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	306	311	303	314
Average Chain Length	2.84	2.77	2.83	2.77
Longest Chain	6	6	7	6
Empty Buckets	61	49	54	55
Chi-squared p-value	0.98	1.00	0.98	1.00
512 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	313	308	298	299
Average Chain Length	2.73	2.79	2.86	2.84
Longest Chain	6	7	7	7
Empty Buckets	54	62	66	61
Chi-squared p-value	1.00	1.00	0.82	0.95

5.2.1 Are these results in line with expectations?

First, I think we can clearly see that the results from the *Bias* Data Set are problematic, particularly when combined with a bucket size of 500 or 512. Leaving that data set aside for now, and considering only Tables 5.1, 5.2 & 5.4 none of the hash functions appear to have performed significantly better than the others. Murmur2, which had such good avalanche results on page 42, seems to have performed on par with the other functions when it comes to distribution and collisions. For example, taking the *Baby Names* data set (Table 5.1) we can see that Murmur2 has the lowest number of empty buckets when distributed among 500 in total, but has a worryingly low p-value in this section. By contrast, when distributed among 512 buckets, Murmur2 actually has the highest number of empty buckets but sees the lowest number of collisions (along with DJBX33A). So there are no clear winners or losers when it comes to distribution and collisions.

Let's briefly compare these results to the expectations from the Poisson Model we outlined in Section 4.2 on page 34. To summarise the results above, Table 5.5 below shows the maximum, minimum and average values for the number of Empty Buckets, Collisions and Chain Length when distributed among 500 buckets (excluding the *Bias* data set).

Table 5.5: Summary of results for 500 buckets

Empty Buckets	Maximum	79
	Average	64
	Minimum	49
Collisions	Maximum	315
	Average	299
	Minimum	277
Longest Chain	Maximum	9
	Average	7.2
	Minimum	6

Recall that expected values from the Poisson Distribution were that we should expect to see 68 empty buckets, 135 buckets with one entry, and 297 buckets with collisions when divided among 500 buckets [1] in total. Hence the average number of collisions and empty buckets observed in Table 5.5 are in line with expectations.

Questions arising from these results

While we may be able to say that the *averages* of our results for three of our four data sets are in line with expectations, that glosses over a number of more interesting outcomes that we see when we look in more detail. We enumerate these below:

1. The most striking result is obviously the complete collapse of the multiplicative hashes (FNV-1, FNV-1a and DJBX33A) when tested using the synthetic bit-string *Bias* data set combined with a non-prime bucket size. See Table 5.3. Here we can see that the every result for DJBX33A is very poor when combined with 500 or 512 buckets. Both of the FNV functions also struggled with 500 and 512 bucket sizes,

[1] Appendix C shows the results for 499 and 512 buckets, both of which also closely matched the approximations.

particularly showing high numbers of empty buckets and extremely low p-values. Murmur2 performed significantly better for this data set.

2. The choice of a prime number seems to have little impact on results for all data sets with the exception of the *Bias* set. Why did this data set struggle so much whereas the others didn't? And was it, in fact, the use of a prime number which made the difference or was it more simply the fact that the number was odd?

3. Also of interest was the apparent disconnect between avalanche performance and distribution, as outlined above. Despite the prowess of the Murmur2 algorithm in avalanche testing, it generally performed no better than the other hashes when tested for distribution and collisions, especially when using the text-based data sets.

4. Do our results imply a uniform distribution? We would expect the number of empty buckets, for example, to follow a normal distribution. If we consider a confidence level of 90%, all fields highlighted in red in the above tables would fail. Even at a confidence level of 95%, not all results would support the null hypothesis. How concerning is this?

5. It would appear that the results of FNV-1a and FNV-1 are identical when run using the *Bias* data set and distributed among 512 buckets. Why is this?

6. In the data set of *IP Addresses* (Table 5.4), the p-values achieved by the "multiplicative hashes" (FNV-1, FNV-1a and DJBX33A) seem suspiciously perfect, achieving between 95% and 100% for all bucket sizes. Why do they perform so consistently well with this data set?

We will explore each of these questions in detail in the following chapter.

Chapter 6

Analysis of Results

In the previous chapter, we presented the results of our avalanche and distribution testing alongside the collision resistance of four hash functions, across four different input types, and three differing bucket sizes. Whilst we saw that the averaged results were roughly in line with expectations, there were a number of interesting outcomes which warrant further investigation. We will explore each of these in detail below.

6.1 Why do FNV-1, FNV-1a and DJBX33A perform so poorly when using the Bias data set?

First, we need to consider the input. The code used to create this data set gives a string of "`fe`" hexadecimal characters, apart from one "`ff`" entry which moves along by one place for each line.

So the first input will start with the following 4 bytes: `fffefefe`, and will continue with `0xfe` bytes to the end of the $1,000$ byte string. The second input will then start with `fefffefe` and will continue with `0xfe` bytes to the end of the $1,000$ byte string. The very last input string will consist of 999 `0xfe`'s before a `0xff` as the final byte.

In binary form, `0xfe` $= 11111110$ while `0xff` $= 11111111$. So almost every byte taken into the hash function is 11111110 in this case.

How does this behave when combined with the hash algorithm?
If we take FNV-1a initially, the algorithm starts with the FNV Offset, which is an odd number. It then XORs this with the first byte. Assuming it starts from the end of the string, this first byte will be 11111110 which will have the effect of flipping each of the bottom 8 bits, **except the last one**. Hence we still have an odd number. This is then multiplied by the FNV Prime, obviously another odd number. This result will be XORed with the next byte, which is again 11111110 and multiplied by the prime again. Hence, until we reach the `0xff` byte somewhere in the string, all steps will result in an odd number. We will then see this switch to an even number whenever the single `0xff` (or 11111111) is XORed in, which will then be maintained into the final result, giving us an even output for every line of input. This means that, when we consider the results mod 500 or mod 512, no odd numbered buckets will ever be filled. This is clear from the results table, as roughly half of the buckets are always left empty.

DJBX33A will suffer from the same general repeating pattern, considering its multiplier is also an odd number. However, the fact that the overall function is significantly more simplistic means that the results are even worse. DJBX33A actually hits every 4th bucket

when distributed among 500 in total. Unlike the FNVs, it is odd-numbered buckets which are filled. However when we use 512 buckets, the outputs are hitting only 16 buckets in total, sending 62/63 results to each, leaving gaps of 32 empty buckets between each filled one and giving a total tally of 496 empty buckets!

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. This is clear from the binary representation of 33: 00100001.

So, as each byte is read in, it is added to the intermediate hash value and then, when multiplied by 33, everything is shifted left by 5 places and the new byte is added on again. This means that the bottom 5 bits are preserved.

In the *Bias* data set, every byte 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.

$$\text{DJBX33A Hash} = (((InitialValue * 33) + Byte_0) * 33 + Byte_1) * 33 + ...$$

To see the bottom 5 bits of this, we consider it mod 32:

$$= ((IV * 33)\text{mod } 32 + Byte_0(\text{mod } 32)) * 33\text{mod } 32 + Byte_1(\text{mod } 32)) * 33\text{mod } 32 + ...$$

$$= ((IV * 1)\text{mod } 32 + Byte_0(\text{mod } 32)) * 1\text{mod } 32 + Byte_1\text{mod } 32)) * 1\text{mod } 32 + ...$$

$$= (IV + Byte_0 + Byte_1 + ...)\text{mod } 32$$

In our case, the Initial Value is zero[1], hence what we see is

$$= (Byte_0 + Byte_1 + Byte_2 + ...)\text{mod } 32$$

Which is why only every 32$^{\text{nd}}$ bucket is filled. The bottom 5 bits are basically the sum of the input bytes mod 32, which is constant for this data set.

The same argument holds for distribution among 500 buckets, however in this case we know that $gcd(500, 32) = 4$, hence it's every 4$^{\text{th}}$ bucket which is filled.

6.2 What effect does the use of a prime number have on distribution and is the fact that it is prime actually the pivotal factor?

First, we must point out that the use of a prime number of buckets appeared to have little effect on the distribution seen in three of our four data sets. The *Bias* data set however made clear that it can offer benefits in breaking the patterns caused by some input types.

In the point above, we investigated why the repeating pattern of the *Bias* data set caused so many difficulties for the multiplicative hashes combined with even numbered buckets. We can see that these hashes all performed better when combined with a bucket size of 499. Is this due to the fact that 499 is a prime number or, in fact, it is simply that it is odd?

To check, we looked at a sample set of all bucket sizes between 496 and 516. 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 6.1 for results.

[1] Although we used an IV of zero, the use of a different constant would not change the effect seen

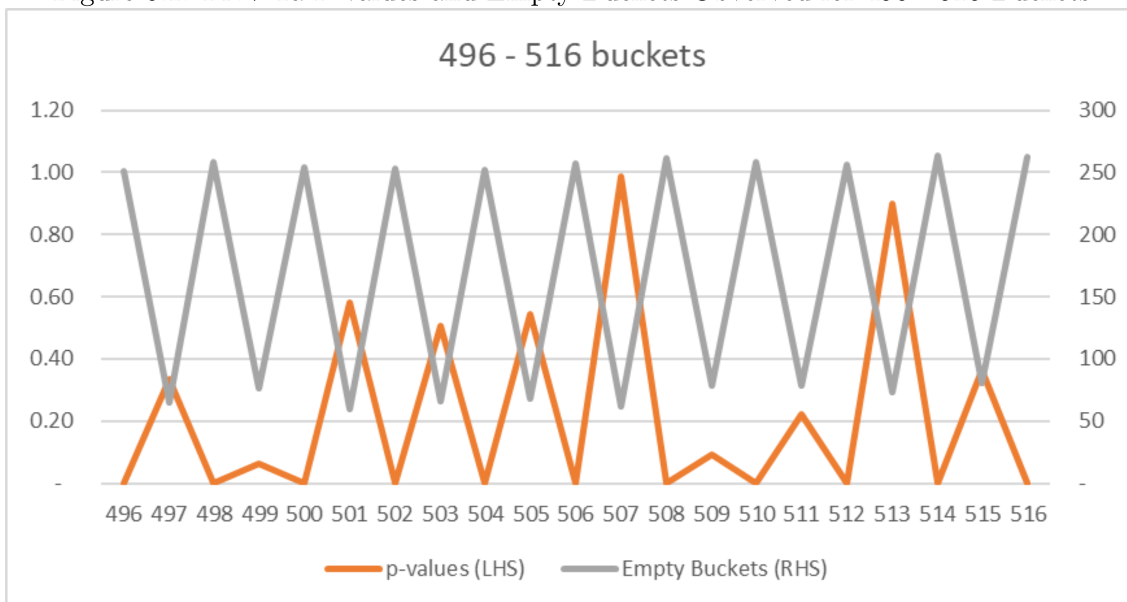
The number of empty buckets (grey line, axis on right hand side) bounces very regularly between a minimum of 60 and a maximum of 263. The Poisson Model would suggest that the number of empty buckets should range from 68 (for 496 total buckets) to 74 (for 516 total buckets). In general however, when the number of buckets is an even number, only even-numbered buckets were filled, meaning that half of these were left empty as explained in Section 6.1 above. Therefore using an even number of buckets results in between 227 and 263 buckets being left empty, while an odd number of buckets sees much better results of 60 to 81 empty buckets.

P-values however tell a 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, the two lowest p-values are seen when using bucket sizes of 499 and 509, both of which are prime numbers!

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 vs power-of-two could be reduced to a simple even vs odd question...

Figure 6.1: FNV-1a P-values and Empty Buckets Obverved for 496 - 516 Buckets



Is there a reason why 507 had the highest p-value while 499 and 509 were so low? This is an outstanding question which could warrant further investigation. One explanation we considered was around the **order** of the group generated by 2 when considered mod 507 or mod 509. As a reminder, the order of a group is the number of elements contained in that group. It can also be considered as the smallest positive integer m such that $a^m = e$ (where e denotes the identity element of the group).

For example, the order of the group generated by 2 is quite different when considered mod 7 or mod 9:

mod 7	mod 9
$2^0 = 1 \pmod 7$	$2^0 = 1 \pmod 9$
$2^1 = 2 \pmod 7$	$2^1 = 2 \pmod 9$
$2^2 = 4 \pmod 7$	$2^2 = 4 \pmod 9$
$2^3 = 1 \pmod 7$	$2^3 = 8 \pmod 9$
	$2^4 = 7 \pmod 9$
	$2^5 = 5 \pmod 9$
	$2^6 = 1 \pmod 9$
Order = 3	**Order = 6**

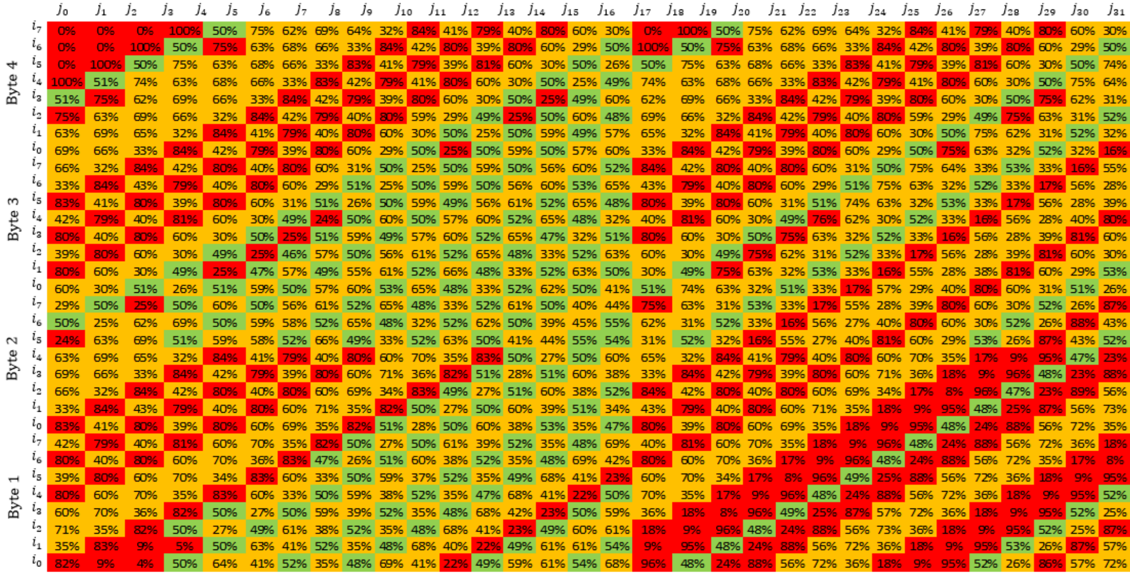
We had wondered if perhaps the group generated by 2 mod 509 would wrap around to repeat values faster than the group generated by 2 mod 507. However that does not appear to be the case, and both groups have unique values all the way to 2^{32}.

6.3 Why are we seeing a disconnect between avalanche performance and distribution?

This was a very interesting result — we had expected that a good avalanche would imply a good distribution. However, when we look into the structure of Murmur2, it becomes

clearer why these are not actually equivalent. Looking at the pseudocode of Murmur2 on page 32, we see that k is initially set to the 4 bytes of data which have been read in. The following three lines of code are invertible mixing steps, which mix the data around to create a more random looking input. It is not until the final two lines of the initial part of the code that h is incorporated and when one looks at this part of the code, it looks very familiar...Step 1: multiply an initialization vector by a prime, Step 2: XOR with the data. It is, in fact, identical to FNV-1! So it could be said that, up to this point, Murmur2 is basically FNV-1 following an additional mixing of input data. How would Murmur2's performance compare if these mixing steps were removed? Graph 6.2 shows Murmur2's Avalanche results with the mixing step removed.

Figure 6.2: Murmur2 with Mixing Step Removed 32 x 32 Avalanche



The difference is quite striking when we compare to the original Murmur2 Avalanche graph on page 42. There are now a significant number of red entries and some interesting patterns emerging. The mixing step is therefore obviously crucial to Murmur2's excellent avalanche performance.

The above graph only considered the removal of the mixing steps at the start of the Murmur2 algorithm. The finalisation step is now the interesting part: how important is this step in Murmur2's performance? As the *Bias* data set was the only one for which Murmur2 produced significantly better distribution and collision results, we examined if the addition of a finalisation step would have improved the performance of FNV-1 on this data set.

Table 6.1 compares the original performance of FNV-1 with the new result of FNV-1 combined with Murmur2's finalisation step, using the *Bias* data-set. We show the results distributed over 500 and 512 buckets, both of which were challenging for FNV-1.

The adjustment immediately brings both the number of collisions and empty buckets back to the levels we would expect for a normal distribution, and the p-values agree with this. I think we can therefore be confident that the finalisation step produces a significant improvement in performance and this could be what makes the difference in

Table 6.1: FNV-1 with Murmur2's Finalisation Step

500 Buckets		
	FNV-1 Original	FNV-1 with Finalisation
Number of Collisions	223	308
Average Chain Length	4.37	2.82
Longest Chain	11	8
Empty Buckets	251	60
Chi-squared p-value	0.00	0.85
512 Buckets		
	FNV-1 Original	FNV-1 with Finalisation
Number of Collisions	256	300
Average Chain Length	3.91	2.89
Longest Chain	4	6
Empty Buckets	256	80
Chi-squared p-value	0.00	0.67

giving Murmur2 those superior distribution and collision results when using the *Bias* data set. However, it is the initial mixing step which creates the winning avalanche performance.

6.4 Can we say with confidence that we see a normal distribution when we look at the distribution of empty buckets?

As we can be quite confident that the *Bias* data set does NOT result in a normal distribution, it is excluded for this analysis.

Why do we expect to see a normal distribution for the three remaining data sets? As discussed briefly in Section 4.2 on Balls and Bins, we can say that, when outcomes are considered to be random and independent, each bucket has the same probability, p, of receiving an output and therefore the same probability of being left empty: $q = 1 - p$. Assuming that each hash output is distributed independently, we can now think of the number of empty buckets we see in terms of a binomial distribution. If we have M total buckets, and q is the probability of an empty bucket, we can then say that

$$\mathbf{P}[\text{seeing k empty buckets}] = \binom{M}{k} q^k (1-q)^{(M-k)}$$

In cases where the sample size is large enough, the Central Limit Theorem tells us that the distribution of the binomial becomes approximately normal. We therefore considered the Cumulative Distribution Function of the number of empty buckets, and compared it to that of a normal distribution. The CDF measures the probability of seeing up to a certain number of buckets:

$$\mathbf{P}[X \leq k] = \sum_{i=1}^{\lfloor k \rfloor} \binom{M}{i} q^i (1-q)^{(M-i)}$$

Considering the *Baby Names, Common Words* and *IP Addresses* data sets, across 499, 500 and 512 buckets, Graph 6.3 shows the CDF of our observed results versus that of a normal distribution. The chart measures the number of empty buckets on the x-axis, starting from 39 (3 standard deviations below the mean) to 92 (3 standard deviations above the mean) with a mean of 66.

As you can see, the distribution of our observed results (grey line) closely follows that of the normal distribution (orange line).

6.4.1 Detailed Statistical Inspection

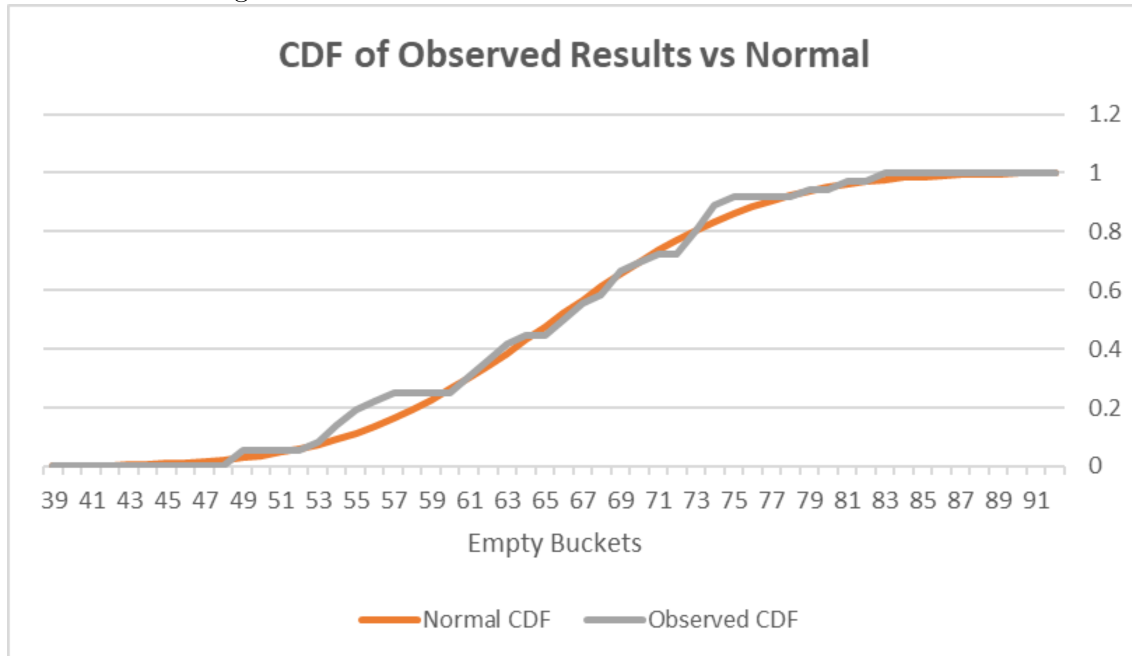
To give us more confidence than just a visual representation, I considered the Anderson-Darling test for normality [29]. This test measures if a given sample of data is drawn from a specific probability distribution, in our case the normal distribution.

$$AD = -n - \frac{1}{n}\sum_{i=1}^{n}(2i-1)[\ln(F(X_i) + \ln(1 - F(X_{n-i+1})]$$

where n = number of data points in the sample, X_i is the ith data sample when sorted from smallest to largest, and $F(X)$ is the cumulative distribution function for the specified distribution.

The result showed an $AD = 0.314$ which is well inside critical values for all significance levels. This statistic is generally used to compute a p-value which, in our case, was 0.53, giving us no reason to doubt that we have a normal distribution.

Figure 6.3: CDF: Observed results vs normal distribution



What about our outlier results?

If we review the results across the *Baby Names, Common Words*, and *IP Addresses* data sets, we can see three p-values which would imply a rejection of the null hypothesis, assuming a confidence level of 95%. Is this enough to make us believe that we do not have a normal distribution?

When conducting a large number of tests simultaneously, there is an increased chance of making a "Type I" error, i.e. rejecting the null hypothesis when you should not. In other words, as the number of tests done increases, so too do the chances of coming across a significant result purely by chance.

In our results above, we can apply a correction method to ascertain if the significant results are still significant when considered in terms of the number of tests being performed.

The **Bonferroni Correction** [30] is used to reduce the chances of making a Type I error. It does this by adjusting the "family-wise-error rate" across our "family" of tests to bring it back to what the individual significance level would be for one test. The family-wise error rate is the probability of making at least one Type I error among ALL the hypothesis tests when performing multiple tests.

When using the Bonferroni Correction, each hypothesis is therefore evaluated at a significance level adjusted for the number of tests, i.e. $\frac{\alpha}{m}$, where α = significance level and m = number of tests.

Considering the three relevant data sets above and using a confidence level of 95%, we therefore need to compare the p-values to $\frac{5\%}{36} = 0.0014$. It can immediately be seen that none of our p-values would be considered significant when using this correction.

Bonferroni Correction is considered to be quite a conservative method, and may be susceptible to Type II errors, i.e. failing to reject the null hypothesis when in fact there are enough significant results to suggest that you should. We therefore also considered

the **Benjamini-Hochberg Procedure** [31] which is used to control the False Discovery Rate (FDR), i.e. the expected proportion of "discoveries" (significant results) which are actually false positives.

With this method, we first choose a value for the FDR: 10% is the lowest which is commonly used, often rates up to 25% are seen. We then find the largest k such that $p_{(k)} \leq \frac{k}{m}.Q$, where p represents the observed p-values, m is the number of tests done, and Q is our chosen FDR.

In practice, we rank the results by their p-value and then compare these p-values to $\frac{Rank}{m}.Q$, which is referred to as the critical value. We find the largest p-value which is less than its critical value and ALL p-values which are ranked above this (i.e. lower p-values / higher rank) are considered significant, regardless of whether their own p-value is less than or greater than their critical value.

We can see in Table 6.2 below that none of our p-values are less than or equal to the Benjamini-Hochberg critical value when using a FDR of 10%. In fact, this also holds true when tested using a FDR of 25% and so we fail to reject the null hypothesis.

Table 6.2: Benjamini-Hochberg Procedure Ranking

p-value	Rank	FDR	BH Correction
0.01	1	10%	0.0028
0.04	2	10%	0.0056
0.04	3	10%	0.0083
0.07	4	10%	0.0111
0.08	5	10%	0.0139
0.14	6	10%	0.0167
0.16	7	10%	0.0194
0.17	8	10%	0.0222
0.20	9	10%	0.0250
0.35	10	10%	0.0278
0.37	11	10%	0.0306
0.45	12	10%	0.0333
0.46	13	10%	0.0361
0.47	14	10%	0.0389
0.52	15	10%	0.0417
0.55	16	10%	0.0444
0.61	17	10%	0.0472
0.65	18	10%	0.0500
0.70	19	10%	0.0528
0.73	20	10%	0.0556
0.75	21	10%	0.0583
0.79	22	10%	0.0611
0.81	23	10%	0.0639
0.94	24	10%	0.0667
0.94	25	10%	0.0694
0.96	26	10%	0.0722
0.98	27	10%	0.0750
0.98	28	10%	0.0778
0.99	29	10%	0.0806
0.99	30	10%	0.0833
1.00	31	10%	0.0861
1.00	32	10%	0.0889
1.00	33	10%	0.0917
1.00	34	10%	0.0944
1.00	35	10%	0.0972
1.00	36	10%	0.1000

6.5 Why are the results for FNV-1 and FNV-1a identical when using the Bias data-set combined with 512 buckets?

As outlined in point 1 above, FNV-1a's starting point is the FNV Offset, which is an odd number. It then XORs this with the first byte, 11111110, the result of which will still be an odd number. This is then multiplied by the FNV Prime, obviously another odd number.

Although FNV-1 does these steps in different order, it still produces an odd number, as the Offset $\times$ Prime will be odd, and then XORing with 11111110 will not change that.

Therefore, it follows the same pattern of only hitting every even-numbered bucket when distributed among either 500 or 512 buckets, giving an identical distribution.

6.6 Why do all hashes except Murmur2 achieve nearly perfect p-values on the IP Addresses data set?

The first possible explanation we considered was that *IP Addresses* is the only data-set which is exactly 4-bytes in length. This would not work quite so well for Murmur2, as this hash takes in data 4 bytes at a time, meaning that the algorithm is only running once. FNV-1, FNV-1a and DJBX33A take in one byte at a time, which means that they run exactly 4 times with the IP addresses as input.

There is also the fact the IP addresses already appear quite random — perhaps this negates the benefit of Murmur2's mixing function? In fact, they appear as a 32-bit random-looking sequence, almost like the output of a hash function! I wonder how these would be distributed without being hashed?

Table 6.3: Distribution and Collision Resistance of UNHASHED IP Addresses

UNHASHED			
	500 buckets	499 buckets	512 buckets
Number of Collisions	280	303	122
Average Chain Length	2.99	2.87	7.57
Longest Chain	7	6	202
Empty Buckets	57	67	315
Chi-squared p-value	0.40	0.90	0.00

As we can see from Table 6.3, the results seen when we distribute the unhashed IP addresses among 499 or 500 buckets are actually quite comparable to the distribution & collision resistance of hashed outputs observed in Section 5.2. Likewise, the p-values seen are very acceptable. However, when combined with 512 buckets, the performance falls apart: not exactly surprising that it should struggle more with the power-of-two bucket size. Perhaps the inputs are not as random as we first thought.

How does it compare to the hash of a Random 4-byte input?

Next, let us see if a truly randomly generated 4-byte input would produce similar results to what we saw when we hashed the *IP Addresses* data set: see Table 6.4

Table 6.4: Distribution and Collision Resistance of Random 4-byte Data

500 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	298	287	300	296
Average Chain Length	2.92	2.98	2.85	2.94
Longest Chain	8	8	8	6
Empty Buckets	73	69	54	73
Chi-squared p-value	0.43	0.04	0.93	0.49
499 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	296	296	284	291
Average Chain Length	2.90	2.88	3.05	2.92
Longest Chain	7	7	9	7
Empty Buckets	60	55	80	57
Chi-squared p-value	0.76	0.85	0.00	0.52
512 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	299	300	305	303
Average Chain Length	2.89	2.90	2.86	2.84
Longest Chain	7	7	7	7
Empty Buckets	78	83	78	70
Chi-squared p-value	0.54	0.33	0.61	0.73

The nearly perfect p-values we observed in the original analysis are not seen when using randomly-generated 4-byte inputs, and the number of empty buckets has increased. So what is it about the structure of IP addresses that makes them behave so well for FNV-1, FNV-1a and DJBX33A?

Sequential Input Data

When examining the data set in more detail, we noticed quite a high frequency of sequential inputs. See Table 6.5 for example. 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? Even without the example of web crawlers, the structure of IP addresses would mean that it could indeed be common to have the higher bits repeating regularly; in basic terms the most significant bits of the IP address represent the assigned network, while the least significant bits the host identifier. It would be quite conceivable for a website to often see a high number of hits from within the same network. 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.

It is clear that FNV-1a suffers from a number of repeating patterns for this sample of inputs. For example, the third from left hexadecimal digit of the output is always 9, the fourth is always *a* and, for all bar two of the results, the first character is *e*. Which shows that the first three bytes of the input being identical has carried through to the top two bytes of the output.

In fact, the pattern persists even as far as the fifth digit of the output, which is always

Table 6.5: Example of Sequential Input lines from IP Addresses Data Set

IP Address	Hexadecimal	FNV-1a Hash	Murmur2 Hash
220.181.108.80	0xdcb56c50	0xe49a38c6	0x162466a3
220.181.108.81	0xdcb56c51	0xe59a3a59	0x8e1d95f7
220.181.108.82	0xdcb56c52	0xe29a35a0	0x527f6193
220.181.108.83	0xdcb56c53	0xe39a3733	0x845d8035
220.181.108.84	0xdcb56c54	0xe89a3f12	0x5dc18a76
220.181.108.85	0xdcb56c55	0xe99a40a5	0x04774fe3
220.181.108.86	0xdcb56c56	0xe69a3bec	0xbc9bd96c
220.181.108.87	0xdcb56c57	0xe79a3d7f	0x98c35aa7
220.181.108.88	0xdcb56c58	0xec9a455e	0xffa835df
220.181.108.89	0xdcb56c59	0xed9a46f1	0x2f46efde
220.181.108.90	0xdcb56c5a	0xea9a4238	0x10e3332b
220.181.108.91	0xdcb56c5b	0xeb9a43cb	0x8c6beb04
220.181.108.92	0xdcb56c5c	0xf09a4baa	0x0cfb95d6
220.181.108.93	0xdcb56c5d	0xf19a4d3d	0x1b446648
220.181.108.94	0xdcb56c5e	0xee9a4884	0xcf9bf437
220.181.108.95	0xdcb56c5f	0xef9a4a17	0xb95ebde4

either 3 or 4. Also of note is that whenever the input is odd, the FNV-1a output is also odd and the same for even inputs. So the final digit of output is influenced by the input too.
When examining the full 1,000 line input to see if the patterns from this sample can be extrapolated to the whole input, we found the following:

1. The data set contains 361 inputs which are part of a sequence, with the sequence lengths ranging from 2 up to 25 lines.

2. The FNV-1a hash of every sequence sees repeating patterns similar to those above. In all hashed sequences, the 3rd and 4th digit remain constant, while the first and fifth digits are chosen from a restricted number of possibilities.
   When we consider the pattern of the FNV prime (reminder: the FNV primes were chosen to have a 1-bit roughly two-thirds of the way up, one at 2^8, and a small number in the bottom 8) this pattern makes some sense. The third and fourth hexadecimal digits appear to fall within the zeros below the topmost 1-bit, the same bit which affects the second, and occasionally the first digit.

3. 81% of sequential inputs see even/even or odd/odd matching between the input and FNV-1a hash. The non-sequenced part of the data set only sees this matching 47% of the time, which is comparable to the result of 49% for the Random 4-byte data set.

So it appears that the sequential inputs are affecting the performance of the FNV-1a hash. To determine if it is definitely due to this pattern of sequences rather than any other characteristic specific to IP addresses, we sourced a second data set, again of 1,000 IP addresses from the log file of the personal web server maintained by Dr. David Malone.

This data set was from 2023 and contained only 124 sequenced inputs, the vast majority of which were no more than 3 lines long.

The distribution of these IP addresses looked very different, see Table 6.6 below.

Table 6.6: Distribution and Collision Resistance for Data Set: New IP Addresses from 2023

500 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	300	289	294	295
Average Chain Length	2.91	2.97	2.95	2.87
Longest Chain	8	7	7	8
Empty Buckets	72	69	73	53
Chi-squared p-value	0.43	0.21	0.29	0.96
499 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	291	294	297	305
Average Chain Length	2.93	2.91	2.88	2.87
Longest Chain	7	8	9	8
Empty Buckets	61	61	57	69
Chi-squared p-value	0.70	0.839	0.48	0.30
512 Buckets				
	FNV-1a	FNV-1	Murmur2	DBJX33A
Number of Collisions	290	284	294	291
Average Chain Length	2.91	2.95	2.88	2.96
Longest Chain	8	8	8	6
Empty Buckets	66	67	66	82
Chi-squared p-value	0.44	0.20	0.69	0.36

So I think we can be confident that the sequences are the cause of the above-average p-values seen in Table 5.4. Why should it be the case that repeating patterns actually improve the distribution?

Why would sequential input improve hash performance?

Let's first think about the format of our input.

The sequential entries have the following form, with each letter representing a byte of data:
$A.B.C.X_0$
$A.B.C.X_1$
$A.B.C.X_2$
$A.B.C.X_3$
.
.
.
Now think about how the multiplicative hashes work in general. We'll use FNV-1a as the example but it can be applied to all.

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

where h = intermediate hash value, b_i = the ith byte and p = FNV Prime.

Each of these operations are invertible, hence we could work out h_i if needed as follows

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

Likewise, b_i could be calculated if we knew h_i and h_{i+1}:

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

So, in general terms, we can say that the hash functions can be represented as follows

$$h_{i+1} = f_{b_i}(h_i)$$

and because we have shown they are invertible, it must be possible to say that

$$h_i = f_{b_i}^{-1}(h_{i+1})$$

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 h_0 = FNV Offset, and assuming that we read b_0, b_1 and b_2 in order as A, B and C as described above, then h_3 (as the intermediate hash value following the input of the 3rd byte) will show identical output for each line of the sequential input.
From the above, we can also say that $h_3 = f_{b_3}^{-1}(h_4)$ where f is a bijection.
Therefore, if we know that h_3 is identical for each line, the only way that h_4 could also be identical is if b_3 were non-distinct.
As we know that each of b_3 are distinct, each h_4 as the final hash output must be too. Even better, the fact that we know the final two hexadecimal digits in particular are distinct, combined with the hash switching from even to odd with each input line, will also aid distribution throughout the hash table. So we can say that the high number of sequences in our original *IP Addresses* data set has actually improved its distribution performance.

6.7 Summary

From a relatively short list of observations, we explored a lot in this chapter, with results sometimes leading us in unexpected directions! Having dug into the mechanics behind the hash functions, we could see why they would not combine well with the *Bias* data set or similar, due to the repeating patterns. We learned why Avalanche and Distribution are not necessarily related (and will explore this more in Section 7.1) and highlighted two points which I had not seen referenced previously: the assertion that an odd number of buckets can produce at least as good results as a prime number, and that highly correlated inputs (sequential IP addresses) can, in fact, improve the uniformity of distribution of the hash outputs. Some of these ideas could be expanded further, with the investigation around odd vs prime buckets remaining an open question for now.

In the next chapter, we leave the specific test results behind and look at some more general ideas which caught our interest while researching and testing.

Chapter 7

Observations on Results, Revelations and Arising Questions

7.1 Is Avalanche really a relevant metric?

The importance of a good avalanche effect is obvious for cryptographic hash functions. As an example, refer to Table 6.5 on page 63 which shows some of the sequential input lines from the *IP Addresses* data set along with their hash values using both FNV-1a and Murmur2. 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 more secure.
However, when we examined the **distribution** pattern of both hashes, there was very little difference in performance for the majority of data sets. Is Avalanche actually a good indicator of high quality non-cryptographic hash functions, or has it migrated from the cryptographic world, and just been generally accepted? We looked at a number of papers to see where references to Avalanche as a criterion for non-cryptographic hashes may have originated.

1. The paper *"Novel Non-cryptographic Hash Functions for Networking and Security Applications on FPGA"* [18] 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 ([32] & [33]) but, strangely, neither paper actually mentions Avalanche.

2. In a paper mentioned previously, *"Performance of the most common non-cryptographic hash functions"* [15], 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*" [34]. 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"

We referenced this point briefly on page 33 where we pointed out that perfect avalanche could be achieved by using simple linear operators.

Valloud goes on to state that

> "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"* [35]. 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. The authors state that

> "Random selection techniques have the advantage to select an unbiased and representative subset of the population."
> "Hash-based selection is a deterministic selection based on the packet content. Therefore it is by definition very likely that the selection is biased"

The stated aim of the paper is as follows:

> "The target quality is to emulate random packet selection with hash-based packet selection".

Hence it's clear to see how the mixing property of avalanche would be important to this specific target, i.e. emulating a random output.

(c) In the third paper, *"Hashing Concepts and the Java Programming Language"* [36], it does state that

> "Two criteria weed out most candidates for a satisfactory Java language library hash function"

and 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".

However, the author 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 which can be achieved by hashing: creating a randomised output, or a well distributed output. Again, avalanche is vital only for those in search of a random result.

Another publication from 1996 also references Avalanche: "*Applied Cryptography*" by Bruce Schneier [37]. These early references to avalanche seem to describe a slightly different effect to what we understand today. In *Applied Cryptography*, when describing the DES (Data Encryption Standard) system, Bruce Schneier references an "expansion permutation" which "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"

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".

   When stating 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 [38] is written by none other than the authors of *"Performance of the most common non-cryptographic hash functions"* referenced in point 2 above, with the same supporting sources.

Conclusion

Based on the above sample of evidence, it would seem that Avalanche is an important criterion, but only for a very specific form of non-cryptographic hashing which targets randomisation rather than distribution. The references which 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 [34]) or only apply to very specific requirements (Robert Uzgalis' paper [36]).

7.2 Choice of bucket size depends on algorithm used

Our investigation in Section 6.2 indicted that, when using FNV-1a with a small sample size, any odd number of buckets would give distribution at least as good as a prime number, while even numbers struggled. However, there may be a different view when it comes to highly randomised algorithms, like Murmur2, or indeed if we somehow had a working random oracle (see description on page 14).

Think of the random-style output of Murmur2, as seen in Avalanche Graph 5.9 on page 42. We know that, as for all hash functions, the output size is always a power of 2 as CPU registers are almost always a power of two in size. (Our tests used the 32-bit version of Murmur2 but 64-bit is also available). Therefore, logically, a highly randomised power-of-two output should distribute nicely among a power-of-two number of buckets. In fact, its distribution would be more uniform than trying to distribute among an odd / prime number, into which it cannot divide evenly.

As an easy example, consider an output of every permutation of just 3 bits that is uniformly distributed. Each bit can be either 0 or 1, so we have 2^3 combinations. Every combination has an equal one-eighth probability of occurring, so our output is totally random and would show perfect avalanche. Traditionally, a prime number would be considered the best way to ensure good distribution. However, our 8 random outputs obviously have no way to divide evenly into, for example, 5 buckets.

This also introduces an interesting concept called *modulo bias* [39]. This is easiest to see when shown in an example:

Say we have 32 random outputs (represented by $[0, 31]$) and wish to distribute them uniformly among 6 buckets. We would therefore consider each output $\bmod 6$. Let's look at how these will end up when distributed $\bmod 6$.

Table 7.1: Distribution of 32 Outputs mod 6

0 mod 6	0	6	12	18	24	30
1 mod 6	1	7	13	19	25	31
2 mod 6	2	8	14	20	26	
3 mod 6	3	9	15	21	27	
4 mod 6	4	10	16	22	28	
5 mod 6	5	11	17	23	29	

Bucket 0 and Bucket 1 have one more entry than the others, due to the modulo distribution. This will always be the case, and will always affect the lower numbers, unless the output divides evenly into the number of buckets. We can also say that the probability of an entry being in Bucket 0 or Bucket 1 is $\frac{6}{32}$ whereas the other 4 buckets have a probability of $\frac{5}{32}$, which does not give us uniform distribution.

In general terms, if we have M buckets, k uniformly distributed outputs (which each have a $\frac{1}{2^n}$ chance of occurring), and k_i outputs map to bucket i to achieve uniformity, we

would like to be able to say that the probability of landing in a particular bucket is

$$\frac{1}{M} = \frac{k_i}{2^n}$$

which would imply that $2^n = k_i M$ However, if we know that M is not a power of two, this creates a contradiction.

We can therefore conclude that the prime / odd number of buckets is only required in cases where patterns need to be broken. For algorithms which already show no discernible pattern, or are highly randomised, a bucket size which is a power-of-two may be the best choice. Hence the choice of bucket size is not a unilateral one, it should be considered in conjunction with the choice of algorithm, which in turn may need to be chosen based on expected input type.

7.3 Generating Collisions for FNV

As previously discussed in Section 4.1 on Collisions (page 33), there are two types of collision. While we have previously focused on collisions within a hash table, here we consider collisions caused by two distinct inputs which produce the same hash output. We asked ourselves how easy it would be generate such a collision and started by considering very small inputs. FNV-1a showed zero collisions when tested using an input of every possible 2-byte permutation ($2^{16} = 65,536$ inputs). How large an input would we need before seeing collisions? Let's look at it theoretically first.

7.3.1 Theory of Collisions

Start with a k-byte input, A, comprising bytes $a_1, a_2, ..., a_k$. Assume we have hashed this and produced a hash digest, called H_A.
We now need to find if there exists a second input, B, made up of bytes $b_1, b_2, ...b_l$ which could also produce the hash output H_A.
Let's assume first that we have hashed the first $(l-1)$ bytes of B and have produced an output, x. Therefore, we hope to hash the final byte, b_l and produce H_A as the final hash output, i.e.

$$f_{b_l}(x) = H_A$$

Setting this function f to be FNV-1a, this becomes

$$(b_l \wedge x) \times p = H_A$$

As discussed in in Section 6.6 (specifically page 65), each of the operations used in FNV-1a are invertible. Therefore, this can be written as

$$b_l = (H_A \times p^{-1}) \wedge x$$

But here we run into a slight problem: We know that x, p and H_A are all 32-bit numbers. However, our byte, b_l, is only 8 bits long. Therefore, we know that the first 24 bits of the right hand side of the above equation must be zeros. If we are willing to assume that x is uniformly distributed over all 2^{32} values, the chance of the first 24 bits being zero is $\frac{1}{2^{24}}$.

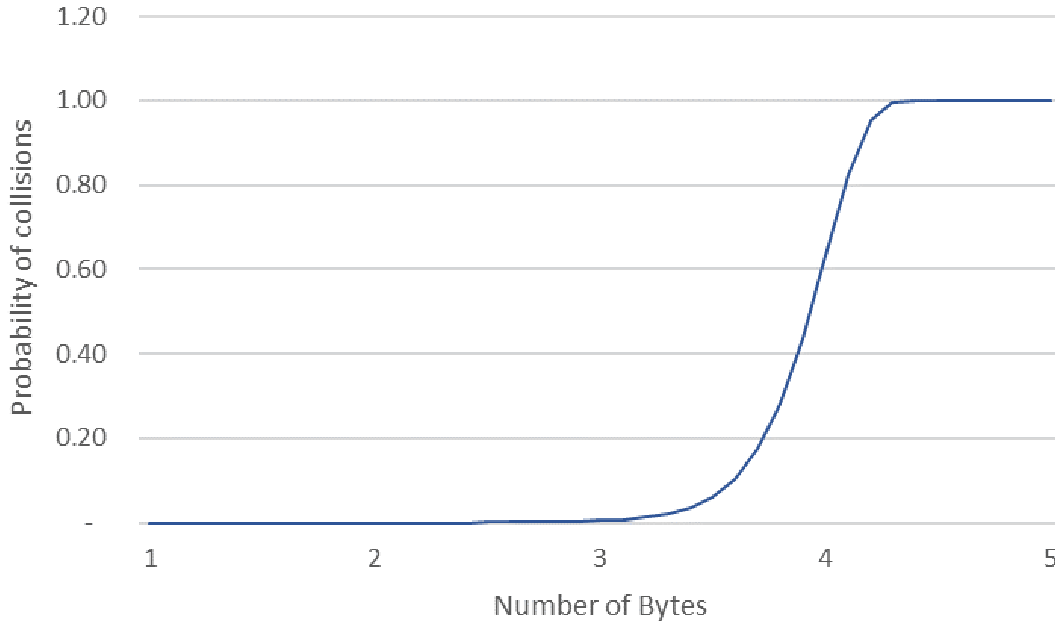
We had the freedom to choose the previous $b_1, b_2, ..., b_{l-1}$ bytes in any way we chose, therefore there are $2^{8(l-1)}$ routes to get here. The chances that none of these combinations

give us the collision we seek is $(1 - \frac{1}{2^{24}})^{2^{8(l-1)}}$ and therefore the probability that we see *at least one* collision is

$$1 - \left(1 - \frac{1}{2^{24}}\right)^{2^{8(l-1)}}$$

It's clear that this will vary significantly based on the value of l. Graph 7.1 below shows how the probability of finding at least one collision moves in line with the number of bytes hashed.

Figure 7.1: Probability of Finding Collisions



The graph shows that the probability of finding a collision when hashing three bytes or less is effectively zero. Increasing to 4 bytes brings this up to approximately 63.212% probability, while we have 100% probability of being able to find at least one collision when we reach 5 bytes.

To get an approximation for where the transition in Figure 7.1 occurs, we could rewrite the above result as

$$1 - \left(1 - \frac{1}{2^{24}}\right)^{\frac{2^{24}}{2^{24}}.2^{8(l-1)}}$$

and when we compare it to the limit

$$\lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^{mx} \to \mathrm{e}^{-x}$$

we can see that, taking $m = 2^{24}$ gives us

$$1 - \left(1 - \frac{1}{2^{24}}\right)^{\frac{2^{24}}{2^{24}}.2^{8(l-1)}} \approx 1 - \mathrm{e}^{-2^{8(l-1)-24}}$$

These formulae can also be extended to larger bit sizes.
If we take n to be the bit size of our hash function, the formulae are adjusted as follows

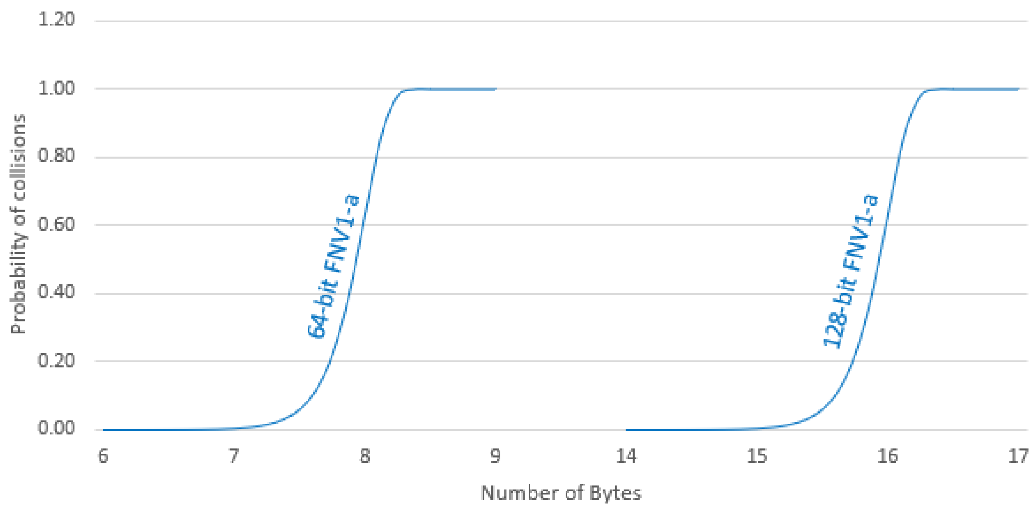
$$1 - \left(1 - \frac{1}{2^{(n-8)}}\right)^{2^{8(l-1)}} \approx 1 - e^{-2^{8(l-1)-(n-8)}}$$

$$\approx 1 - e^{-2^{8l-n}}$$

This equates to $(1 - e^{-1}) = 0.63212$ at exactly the points where $8l = n$. If you recall that l is our number of input bytes, while n is the bit-size of the output hash, this means that collisions can only occur where input and output size match.

When we look at the graph of the results for the 64-bit and 128-bit versions of FNV-1a, we see the same pattern again.

Figure 7.2: Probability of collisions for 64-bit and 128-bit FNV-1a



If we summarise with a small table of results, the pattern is clear to see.

Table 7.2: Probabilities of collisions by input size

Probabilities of collisions by input size			
Approximate Probability	32-bit	64-bit	128-bit
0%	3	7	15
63%	4	8	16
100%	5	9	17

Based on this data, you would expect that the 256-bit hash would have a 63% probability of finding a collision with an input size of 32 bytes, and when checked using our method, that proves to be true.

Link to Zero Hash Challenges?

If we compare these results to what the "Zero-hash challenges" described on page 26 are finding, we saw that the shortest binary data set for which the 32-bit FNV-1a hash is zero has been found to be of length four, while the shortest binary data set for which the 64-bit FNV-1a hash is zero is of length 8. This clearly matches well with what we saw in Figures 7.1 & 7.2 and Table 7.2 above.
When we think about it, if we can generate collisions by choosing the final byte to match an existing hash output, why should this output not be set to zero using the same calculations? i.e. finding a collision and finding a zero hash should take equal work.
No results have yet been found for the 128-bit FNV-1a hash zero challenge but, based on our findings above, it seems likely that that the shortest data set to produce a zero result would be 16 bytes in length.

7.3.2 Considering Collisions More Generally

In this section, we will look at the idealised scenario — almost as though we had a random oracle (see Section 2.2). We can therefore employ the Poisson Model which we introduced in Section 4.2 but a special case thereof: as we ascertained above that we only expect collisions when the input size is the same as the output size, we can consider this a case of the Poisson Distribution where the mean, λ, is set to 1.

Recall that the probability of a bucket containing k entries is given by:

$$\mathbf{P}(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

In this case, given that the mean, $\lambda = 1$, the results of the Poisson distribution would look as follows

$$\mathbf{P}(X = 0) = 0.37 \textit{ (empty buckets)}$$

$$\mathbf{P}(X = 1) = 0.37 \textit{ (one entry)}$$

$$\mathbf{P}(X \geq 2) = 1 - \mathbf{P}(X = 0) - \mathbf{P}(X = 1) = 0.26 \textit{ (collisions)}$$

Do our findings match the theory?

In a word, no.

In order to compare to the theory laid out above, we created inputs comprising every possible permutation of 1-byte, 2-byte, 3-byte and 4-byte strings and ran these through each of our algorithms. We measured how many distinct outputs were generated by each, how many outputs were observed twice, three time, four times and so on. Where the number of outputs generated was less than the number of inputs, we obviously see both collisions and the equivalent of "empty buckets". The size of our input sets grew large very quickly; while all three-byte inputs total 16,777,216, when we increase to 4-byte inputs we reach 4,294,967,296 possibilities.

FNV Collision Resistance

Based on our findings in Section 7.3.1, we would not expect to see any collisions generated by FNV-1a before we reach 4 bytes of input. In practice, FNV-1 and FNV-1a both performed perfectly up to 3-byte inputs, seeing zero collisions. However, when we increased to a 4-byte input, we saw only 1,925,392,640 distinct outputs generated for both FNV-1

and FNV-1a, meaning that 2,369,574,656 potential outputs had been not been hit (the equivalent of seeing empty buckets). We also saw more collisions than expected, see table 7.3 below.

Table 7.3: FNV-1 and FNV-1a: Distribution of all possible 4-byte inputs

Distribution of all possible 4-byte inputs			
	FNV-1	FNV-1a	Poisson Indication
# Generated Outputs	1,925,392,640 (45%)	1,925,392,640 (45%)	2,714,937,127 (63%)
#"Empty Buckets"	2,369,574,656 (55%)	2,369,574,656 (55%)	1,580,030,169 (37%)
# "One entry bucket"	532,860,928 (12%)	532,860,928 (12%)	1,580,030,168 (37%)
# Collisions	1,392,531,712 (32%)	1,392,531,712 (32%)	1,134,906,959 (26%)

We see therefore that reality has given us significantly more empty buckets than Poisson would indicate, and slightly more collisions, whereas the number of outputs being generated a single time is only 12% compared to our expectation of 37%. We can also note that the results for FNV-1 and FNV-1a are identical, highlighting that there is no "better" algorithm between the two, when it is not data-dependent.

Murmur Collision Resistance

While Murmur2 also performed perfectly up to three bytes with no collisions, it also saw zero collisions when we reached the 4-byte inputs, which is in fact even further away from our Poisson estimations! The fact that Murmur2 is the only function which reads inputs in a 4-byte block means that it performs somewhat differently (recall pseudocode on page 32). When running on a 4-byte input, the algorithm has therefore run an input step only once. We know that each of the operations within Murmur2 are invertible meaning that the algorithm is a bijection, giving one distinct output for each input. It does also mean that Murmur2 could be susceptible to malicious attacks, as one can work backwards through each step allowing a chosen hash function to be generated once we have the freedom to choose the final 4 input bytes as described in Section 7.3.1.

In an interesting twist, we did find that Murmur2 recorded collisions *between* 1-byte and 2-byte strings. Recall that when reading in less than 4 bytes, the entire mixing step plus the introduction of the hash is skipped meaning that any input of less than 4 bytes can rely on nothing more than the finalisation step. This highlights that Murmur2 may be unsuitable for very short input lengths.

We wondered if perhaps Murmur3 had been adapted to solve for this issue. Murmur3 [25] does actually revert to its originally planned mix of multiplication and rotation in the mixing step. However the overall structure remains broadly similar:

- Set Hash to Seed value
- Read in 4 bytes
- Mix these 4 bytes by a combination of multiplication and rotation
- XOR the mixed input bytes with Hash, then rotate, multiply and add to this.
- XOR in any remaining (< 4) bytes and run one more multiplication / rotation / XOR step.

- The finalisation step (see Table 7.4 below) comprises XOR and multiplication functions on this Hash.

When testing Murmur3 with the same inputs as Murmur2, we noticed that it did not result in the same collisions between different length input strings, and again all outputs were distinct up to 4-byte inputs.

Therefore, for input lengths of less than 4 bytes which are only operated on by the finalisation step, performance would be significantly improved by moving from Murmur2 to Murmur3, despite still relying on generally the same operations. See Table 7.4 below for comparison of finalisation steps, and Table 7.5 for general performance comparison.

Table 7.4: Finalisation Step Comparisons

Finalisation Steps	
Murmur2	Murmur3
hash:= hash XOR (hash >> 13) hash:= hash * 0x5bd1e995 hash:= hash XOR (hash >> 15)	hash:= hash XOR (hash >> 16) hash:= hash * 0x85ebca6b hash:= hash XOR (hash >> 13) hash:= hash * 0xc2b2ae35 hash:= hash XOR (hash >> 16)

Table 7.5: Murmur2 & Murmur3: Distribution of all possible 4-byte inputs

Distribution of all possible 4-byte inputs			
	Murmur2	Murmur3	Poisson Indication
# Generated Outputs	4,294,967,296 (100%)	4,294,967,296 (100%)	2,714,937,127 (63%)
#“Empty Buckets”	0	0	1,580,030,169 (37%)
# “One entry bucket”	4,294,967,296 (100%)	4,294,967,296 (100%)	1,580,030,168 (37%)
# Collisions	0	0	1,134,906,959 (26%)

DJBX33A Collision Resistance

As we have come to expect, DJBX33A showed a significantly worse performance than its peers. Collisions appeared when hashing inputs of just 2-bytes in length, with only 66 outputs being generated once, and 8,671 outputs in collision, out of a total potential number of outputs of 65,536. This number of 66 distinct outputs appearing only once also carried through to 3-byte inputs AND 4-byte. See Table 7.7 below for details.

Table 7.6: DJBX33A: Distribution of all possible 4-byte inputs

Distribution of all possible 4-byte inputs		
	DJBX33A	Poisson Indication
# Generated Outputs	9,450,301 (0.2%)	2,714,937,127 (63%)
#“Empty Buckets”	4,285,516,995 (99.8%)	1,580,030,169 (37%)
# “One entry bucket”	66 (0.000002%)	1,580,030,168 (37%)
# Collisions	9,450,235 (0.2%)	1,134,906,959 (26%)

Table 7.7: Distribution of DJBX33A for all possible 2-,3-, and 4-byte inputs

Distribution of all possible 4-byte inputs		
		DJBX33A
2-byte	# Generated Outputs	8,671
	# Single Output	**66**
	# Collisions	8,605
3-byte	# Generated Outputs	286,366
	# Single Output	**66**
	# Collisions	286,300
4-byte	# Generated Outputs	9,450,301
	# Single Output	**66**
	# Collisions	9,450,235

We explore the reasons behind this pattern in Appendix D.

SHA Cryptographic Hash Collision Resistance

Before we start wondering if perhaps our Poisson assumptions were incorrect, we ran an SHA cryptographic hash, truncated to 32-bits, to see how that would perform for our 4-byte input data:

Table 7.8: Distribution of SHA Compared to Poisson Estimates

Distribution of all possible 4-byte inputs		
	SHA 32-bit	Poisson
# Generated Outputs	2,714,902,107 (63%)	2,714,937,127 (63%)
#"Empty Buckets"	1,580,065,189 (37%)	1,580,030,169 (37%)
# Distinct Output	1,579,979,621 (37%)	1,580,030,168 (37%)
# Collisions	1,134,922,486 (26%)	1,134,906,959 (26%)

So we can be quite confident that the ratios indicated by Poisson are correct. This highlights the fact that, while all of our functions performed reasonably well with small input sizes, more rigorous testing shows up the shortfalls of non-cryptographic functions versus their cryptographic counterparts.

7.4 Summary

As a departure from the usual suite of tests applied to non-cryptographic hash functions, I think this chapter has highlighted some really interesting points. The research quoted on Avalanche seems supportive of our idea that perhaps this metric is not actually strictly relevant to the non-cryptographic world — a debate on this point would be welcome. We touched on the idea of modulo bias and how, in certain circumstances, a power-of-two bucket size is in fact the best for uniform distribution. Finally, our work on "pure collisions" unearthed some interesting results about when we should expect collisions to appear. Our final tests on collision resistance showed the shortfalls of each of our hash functions when compared to both the ideal (as measured by the Poisson model) and a cryptographic hash function.

Chapter 8

Conclusion

8.1 My personal conclusions

Considering that I started with a Google search for "what are hash functions?" I am very excited by the interesting and unexpected results we have uncovered during this thesis. But let's start from the beginning.

The area of cryptographic hash functions and cryptography in general is fascinating. Learning how this world interacts with so many daily tasks, from online shopping to internet downloads made it very relevant. When I then included non-cryptographic hash functions in my research, I was impressed by the deceptive simplicity of structures such as hash tables, and the various ways to manage collisions. In this thesis, we only touched on two methods: separate chaining and open addressing, but there are many versions of open addressing available such as "cuckoo hashing" and "quadratic probing", for example.

It was very exciting to speak with Mr. Landon Curt Noll to gain insight into the creation of the FNV hashes, and understand why each constant / specification was chosen as it was. For example, research alone would never have explained to us why the number of 1 bits in the bottom byte of the FNV prime should be either 4 or 5, and why the restriction of p mod $(2^{40} - 2^{24} - 1) > (2^{24} + 2^{8} + 2^{7})$ was necessary. His help and insight really helped our understanding, not only of FNV, but hash structures in general.

After researching for a number of months, it was exciting to reach the testing phase. Although it seems mundane, the choice of criteria was important and we took a while to ensure we chose the right methods and input. Taking time to consider the maths behind what we were testing was also important, such as Section 4.2 showing Poisson estimations for the number of collisions we should see, and Section 2.3.3 where we examined the link between empty buckets and distribution. We also looked at Avalanche results byte-by-byte (see Subsection 5.1.1) in order to be able to understand why the patterns we saw were appearing. Throughout the thesis as a whole, I enjoyed using various aspects of mathematics such as probability theory and modular arithmetic which had previously been much more theoretical.

Things got really interesting once we had the results of all the tests in. Considering the widespread use of each of the hash functions we tested, I had expected all them to perform quite well. And, overall, that does hold true.

The Avalanche testing, thanks to Bret Mulvey's code [2], takes a large quantity of data and very neatly puts it into an easily-read matrix format. Seeing these patterns emerge so clearly gave us a "roadmap" of sorts to follow the function through and learn why it produces these results. While FNV's one-byte Avalanche graph was not particularly inspiring, by the time we had 4 bytes of data it was actually looking quite impressive.

DJBX33A's relative lack of improvement when more input data was added gave us a hint that the algorithm may be poor, whereas Murmur2's perfect Avalanche performance probably convinced me too strongly that this function would have fantastic results across the board.

When restricted to real-world data sets, the distribution and collision resistance of each of the four hash functions tested were perfectly adequate. That said, I was in for a shock right away when Murmur2 saw such a low p-value on our very first test table on page 45. It quickly became apparent that Murmur2's collision resistance was no better than FNV's and, in fact, its distribution (as measured by p-value) was slightly worse. Likewise, despite its obvious limitations in relation to Avalanche, DJBX33A's distribution and collision resistance results were in line with the others.

Obviously we wanted a more rigorous test and ideally some inputs / combinations which *would* throw up problems, and we certainly saw this with the *Bias* data set. While we had expected that the functions would struggle with what is, in fairness, a difficult data set, I hadn't expected the results to be quite so bad. It also highlighted to me that the question of using a prime number or a power of two as a bucket size is relevant only to specific cases. What really intrigued me were the results seen when using an odd numbered bucket size versus a prime — that was a wholly unexpected outcome with no real explanation yet ... definitely one which merits further work. Another unexpected tangential discovery was when we investigated the reason behind the "suspiciously perfect" p-values seen for FNV and DJBX33A with the *IP Addresses* input data set. Upon first noticing the sequential IP addresses, I did not expect to conclude that such an anomaly can actually result in excellent distribution!

The final section of my thesis was a little sobering. It showed that the test results I had spent so much time investigating were, in fact, such a small subset that it made life "too easy" for the hash functions to perform well. I was becoming proud of my functions, and how well they had performed, until I saw the reality of more rigorous testing (and still on just 4-bytes of input) and the comparison to a cryptographic hash. However, we must remember that they are specifically designed **not** to compete with such functions, and their use is targeted more at the sort of work we considered.

8.2 Discoveries

1. Our first discovery must be to confirm that all four hash functions provide good distribution and collision resistance when combined with real-world data input, and so are very well suited to the job they were designed to do.

2. I believe we have made a solid case for the re-evaluation of Avalanche testing as a tool for measuring non-cryptographic hash functions. We showed that, for the majority of data input sets, an excellent avalanche performance is no indication of good distribution or collision resistance. This can be clearly seen in the tables in Section 5.2. We also investigated some sources of support for Avalanche as a relevant metric in Section 7.1 and found very little evidence to support its use outside the cryptographic world.

3. For inputs with poor distribution, rather than deciding that a chosen hash function is not up to standard, its performance may be improved by combining with either an odd / prime number of buckets (as evidenced by the superior performance of 499 buckets in our results for the *Bias* data set on page 47), or by the use of an

additional finalisation step (see Table 6.1 showing the improvement in performance of FNV-1 when combined with finalisation step in Section 6.3).

4. I believe that we have asked an important question in whether an odd number of buckets may perform as well as a prime number, when combined with certain hash functions (see Section 6.2)

5. Based on what we saw on "pure collisions" in Section 7.3, I believe our testing actually portrays our four tested functions in a somewhat flattering light, thanks to a relatively small input size. Considering the maximum size of our input data sets was less than 2^{10}, which is then mapped to a domain space of size 2^{32}, it was quite easy for collisions to be avoided, meaning that we only saw collisions due to bucket distribution.

6. Certain patterns in the input data (see Section 6.6 for description of sequential inputs and their effects) can actually work in your favour when using hash functions which display repeating patterns and don't prioritise randomisation.

8.3 Further Work

1. There is definitely scope for further work on the optimal choice of bucket size, examining the behaviour of "prime vs odd" numbers for varying input data sets and functions, over a larger sample size.

2. Sample size itself is an interesting topic. As mentioned in point 4 above, the small size of our input data sets means that we were able to avoid all collisions of outputs (as opposed to bucket collisions). Referring back to Section 7.3.2, we saw a rate of collisions for FNV-1 and FNV-1a of 32% when tested using all possible 4-byte combinations. However we saw zero collisions when input size was kept to permutations of 3 bytes (16,744,574 input lines). If we continued to test larger input sizes, would we reach a point where it becomes infeasible or impractical to use FNV hashes or similar?

3. Having suggested in point 2 of Section 8.2 above that FNV-1 (for example) could be improved by the addition of a finalisation step when using difficult data, at what point is the data "difficult enough" to warrant this extra work? If we recall the "sticky state at zero" mentioned by Landon Curt Noll during our interview (see Section 3.2), could the addition of a finalisation step be an potential option to overcome this?

8.4 Summary and farewell

I'm delighted to end this thesis having both answered some questions and found more to pose. I learned a huge amount and not always in the areas I had anticipated — sometimes the tangents are more fun than the main route! All that remains now is for me to thank you, the reader, for having stayed with me until the end. I hope you too found something to interest and excite you.

Appendix A

Convolution

In general terms, a convolution is an operation done on two functions, f and g, which measures the effect of one on the other. It is defined as an integral over the two functions, after one of them has been reversed and shifted. It can be thought of as the effect of moving the function g across the function f, and measuring what shape is formed as they move along t; hence it is a type of integral transform.

It is denoted by $(f * g)(t)$ and is equal to

$$\int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

where τ acts as a dummy variable, allowing the function g to be reversed and then combined with t to move along the τ-axis. As t goes from $-\infty$ to $+\infty$, the integral is measured where $f(\tau)$ and $g(t-\tau)$ intersect.
In the case of discrete convolution, where f and g are sequences defined on the set $\mathbb{Z}$ of integers, convolution can be rewritten as

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

As in a Cauchy product, when the sequences are polynomials then the coefficients of the ordinary product of the polynomials are actually the convolution of the original two sequences, i.e.

$$\left(\sum_{i=0}^{\infty} a_i x^i\right) \cdot \left(\sum_{j=0}^{\infty} b_j x^j\right) = \sum_{k=0}^{\infty} c_k x^k, \text{where } c_k = \sum_{l=0}^{k} a_l b_{k-l}$$

Fast convolution algorithms such as Fast Fourier Transforms (FFTs)[1] can be used for computation of discrete convolutions. These reduce the cost of the convolution from $O(N^2)$ to $O(N \log N)$ complexity for periodic functions with period N.

[1]The circular convolution of two finite sequences can be found by taking an FFT of each, multiplying pointwise and then performing an inverse FFT. A FFT computes the discrete Fourier transform (DFT) of a sequence rapidly by factorizing the DFT matrix into a product of sparse factors.

Appendix B

Avalanche Graphs using Data Set Inputs

The following graphs show the Avalanche effect measured using our chosen input data sets. As you can see, the patterns broadly mirror those seen when using random input data. For example, FNV-1 still has the "red zone" across the top where the latest bytes were read in (see Figure B.1). Both FNVs still show the saw-tooth pattern on the left. Murmur still shows the strongest avalanche effect, but struggles with short inputs which is particularly evident with the *Baby Names* data set (Figure B.9). DJBX33A continues to struggle across the board, but shows a marked improvement in the longer real text input of *Common Words* (Figure B.14).

The *Baby Names* data set has inputs ranging from 3 bytes in length to 11 bytes, hence our matrix is an 88×32 output. *Common Words* stretches to 126 characters (or bytes) at its longest, hence the matrix can only really be seen as a colour representation. The *Bias* data set is our longest, with 1,000 bytes per input line, hence our matrices show only the first 12 bytes, and the final 12 bytes. Finally, *IP Addresses* brings us back to a nice even 32×32 matrix with its 4-byte pattern.

We can also see the benefit of reading data from the end of the input (see explanation on page 38) when we compare inputs of varying length.

Figure B.1: Avalanche Matrix: FNV-1 with Baby Names DataSet

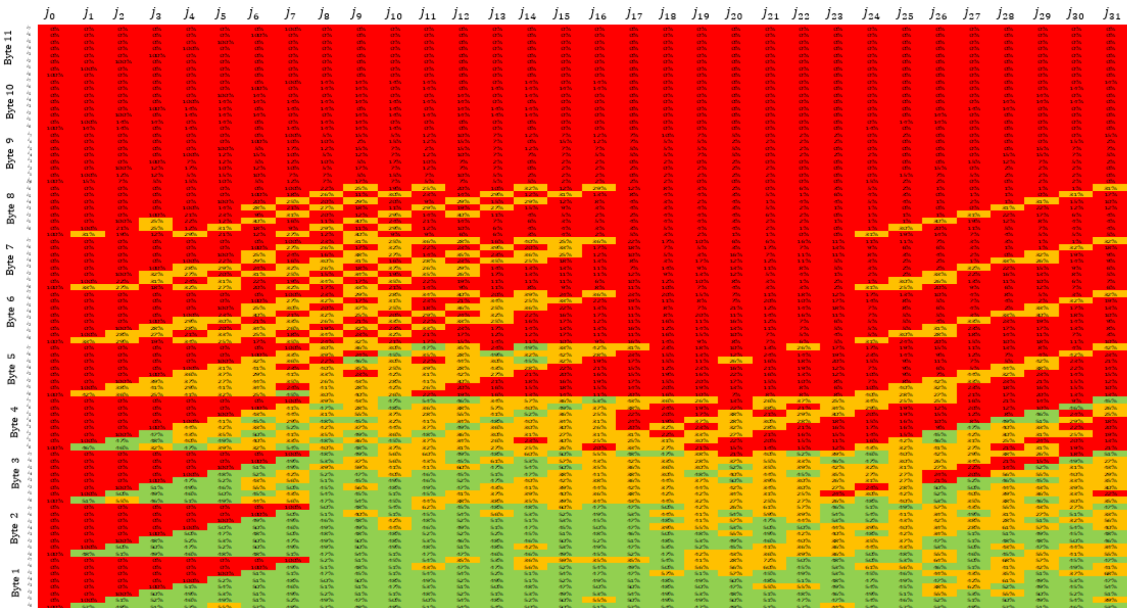


Figure B.2: Avalanche Matrix: FNV-1 with Common Words DataSet

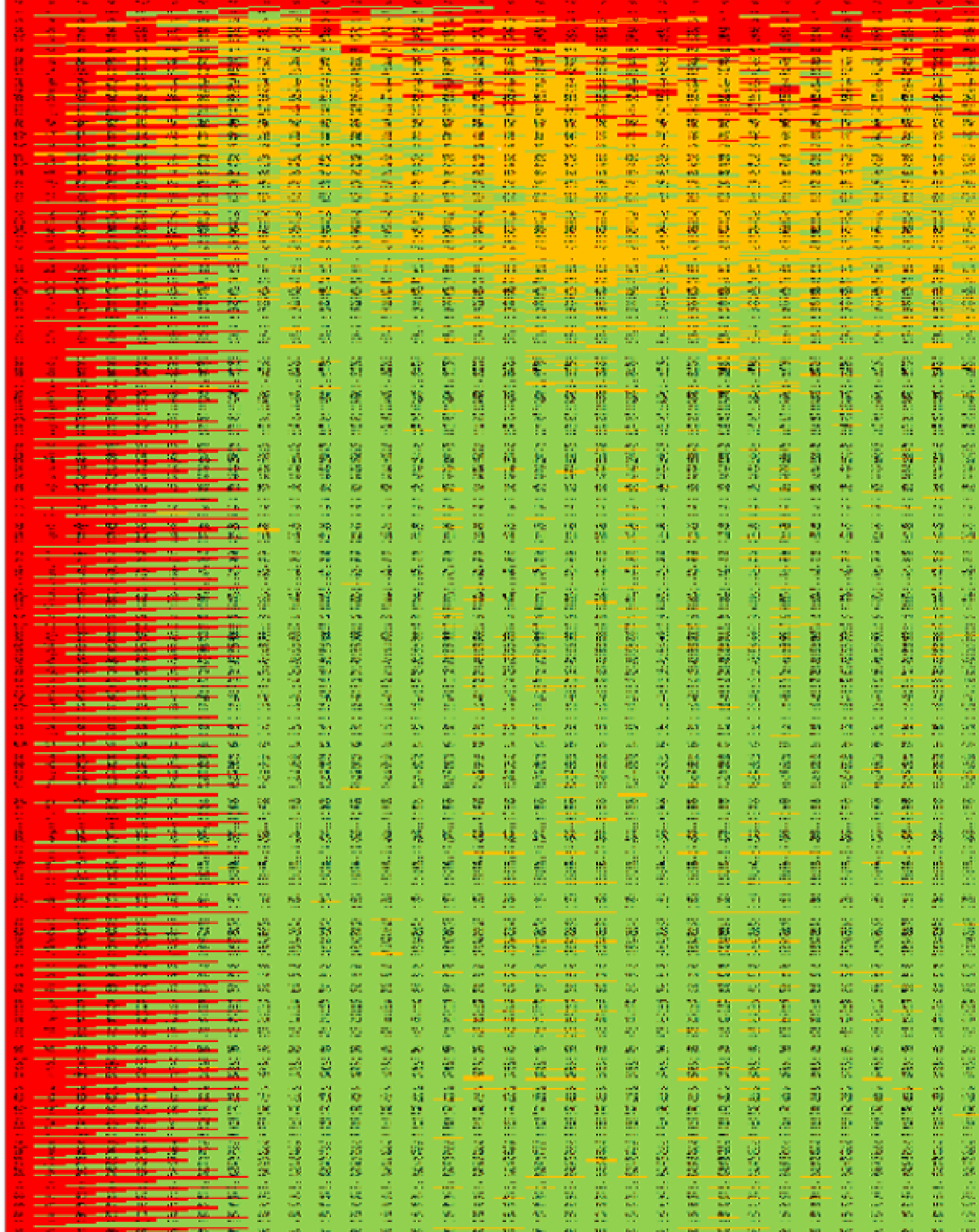


Figure B.3: Avalanche Matrix: FNV-1 with Bias DataSet

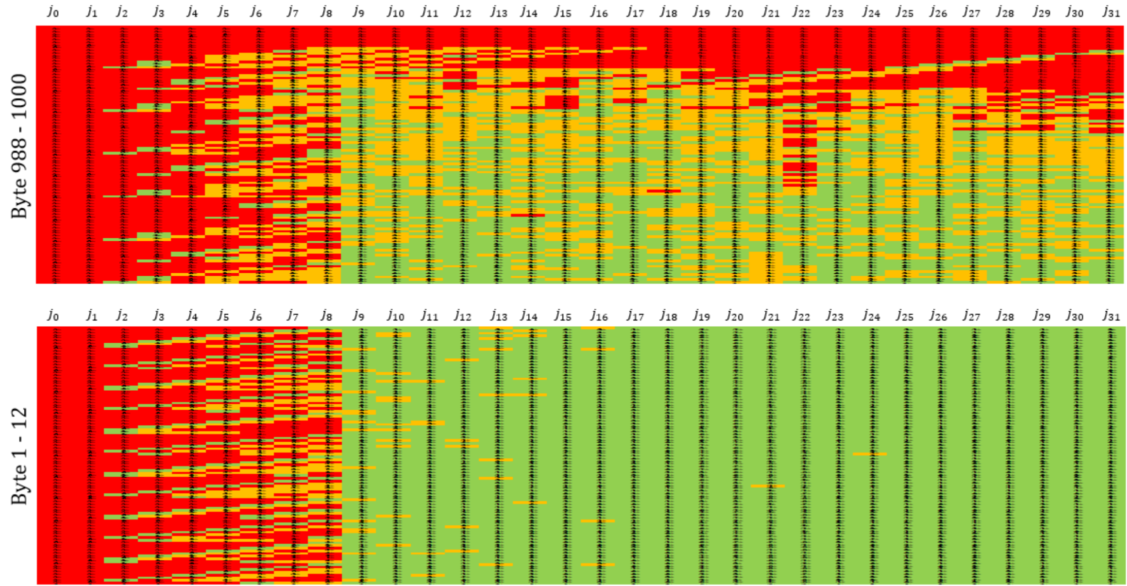


Figure B.4: Avalanche Matrix: FNV-1 with IP Addresses DataSet

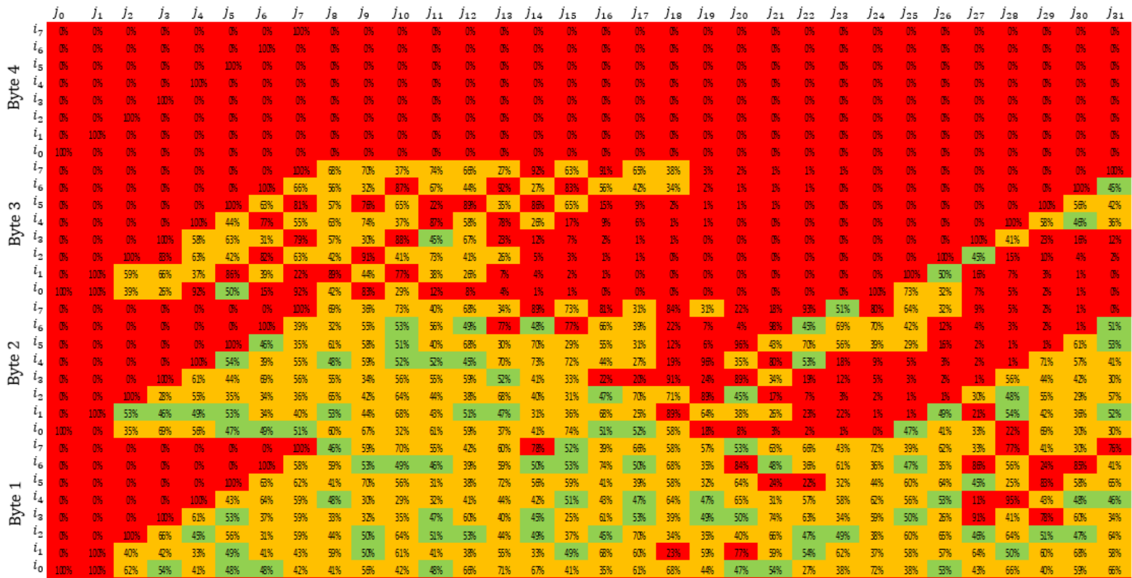


Figure B.5: Avalanche Matrix: FNV-1a with Baby Names DataSet

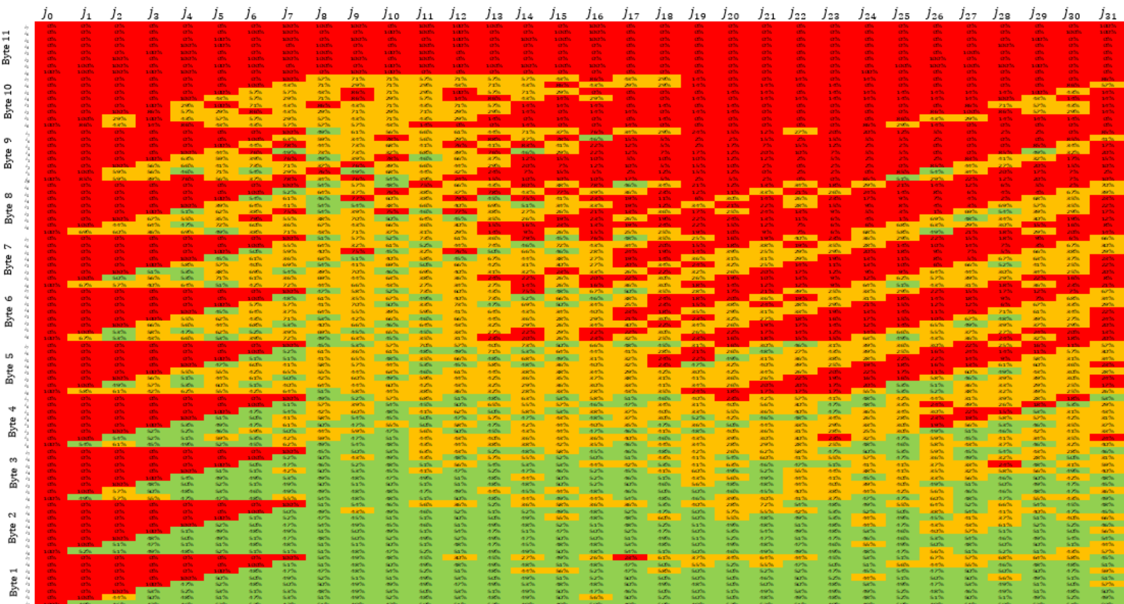


Figure B.6: Avalanche Matrix: FNV-1a with Common Words DataSet

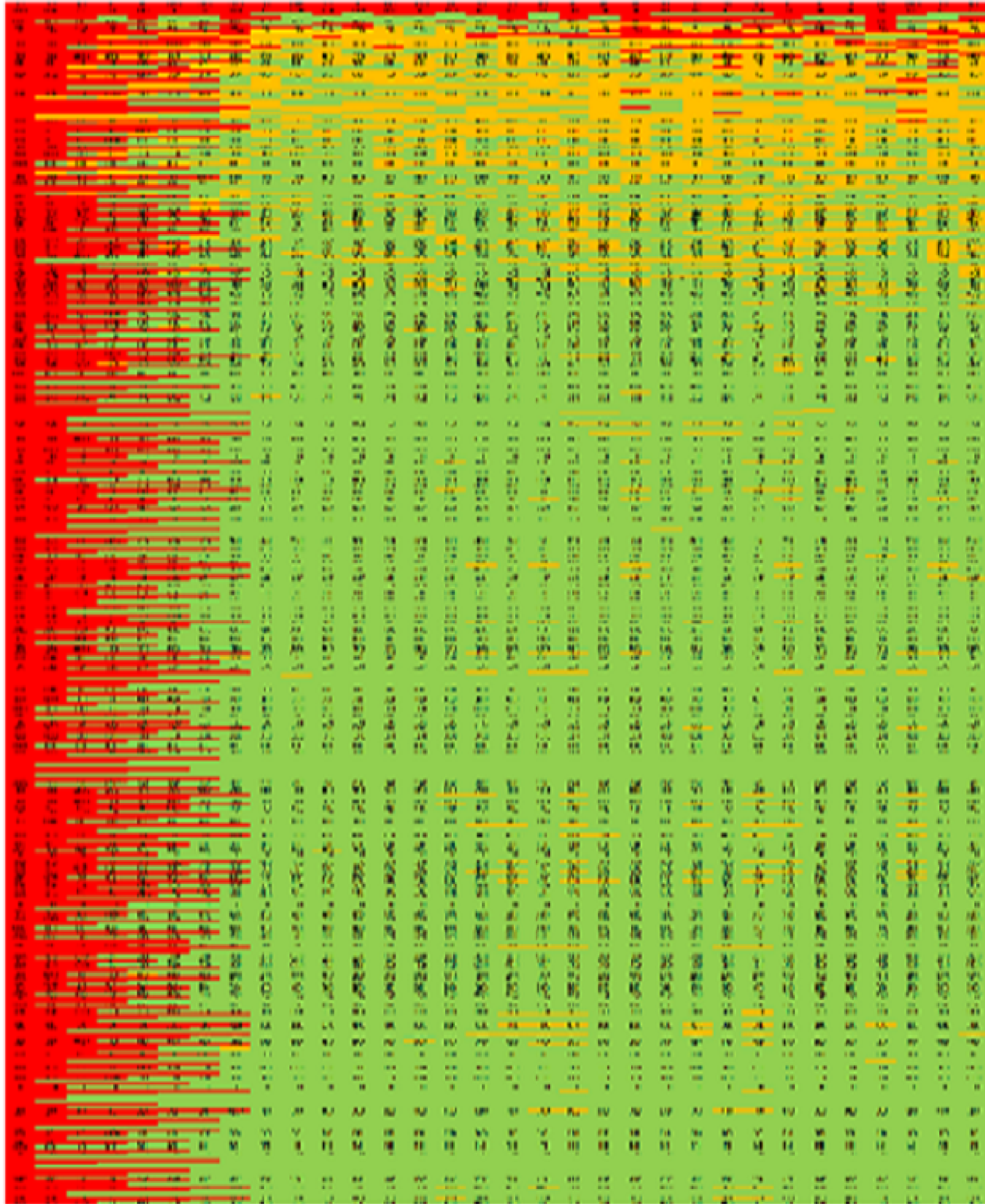


Figure B.7: Avalanche Matrix: FNV-1a with Bias DataSet

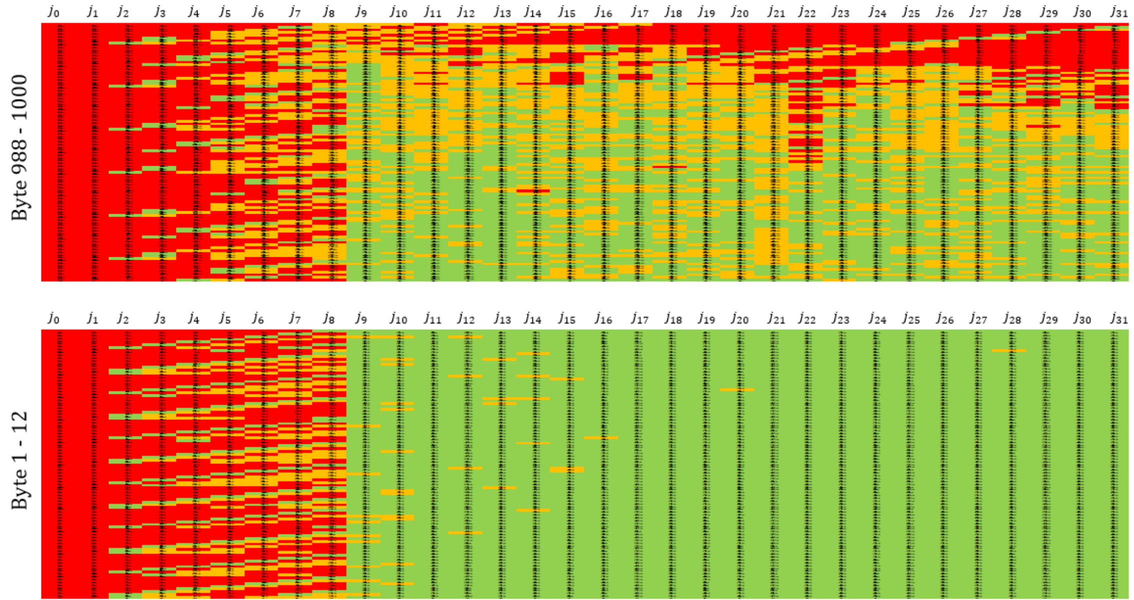


Figure B.8: Avalanche Matrix: FNV-1a with IP Addresses DataSet

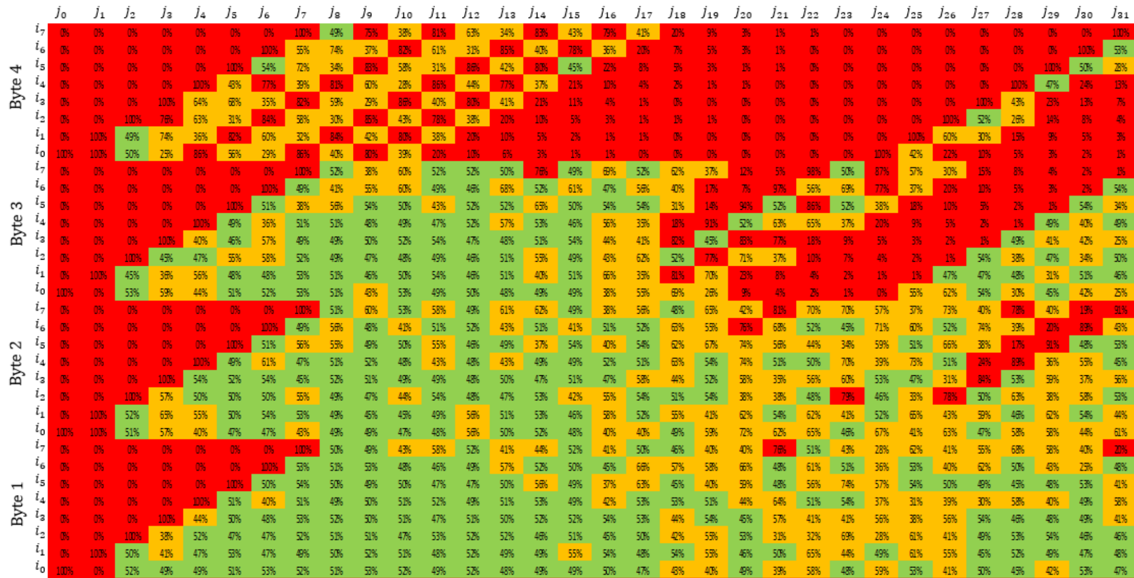


Figure B.9: Avalanche Matrix: Murmur2 with Baby Names DataSet

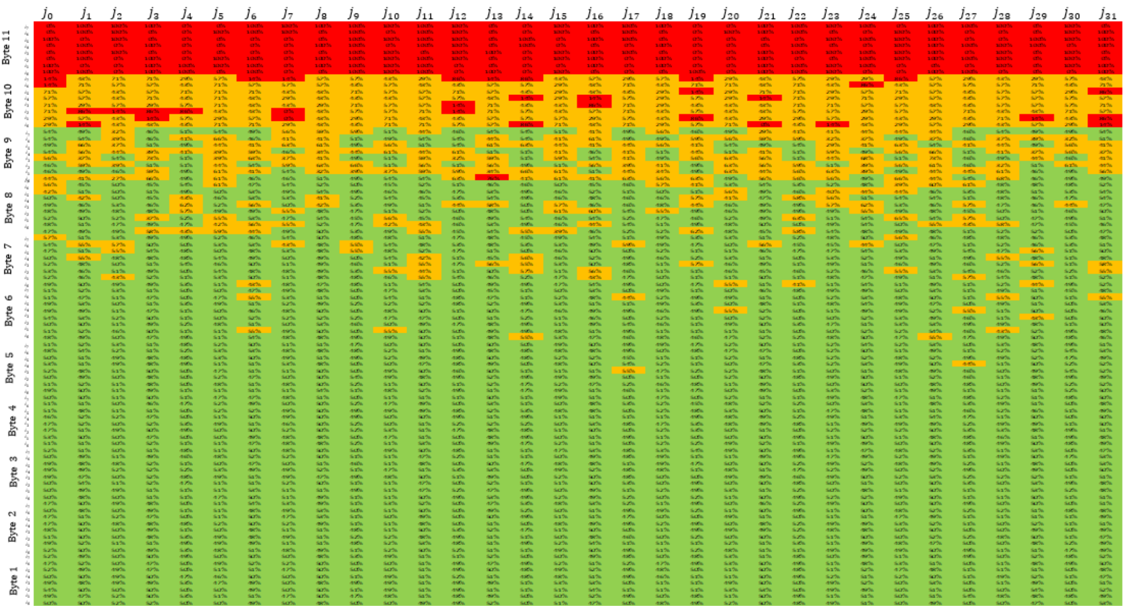


Figure B.10: Avalanche Matrix: Murmur2 with Common Words DataSet

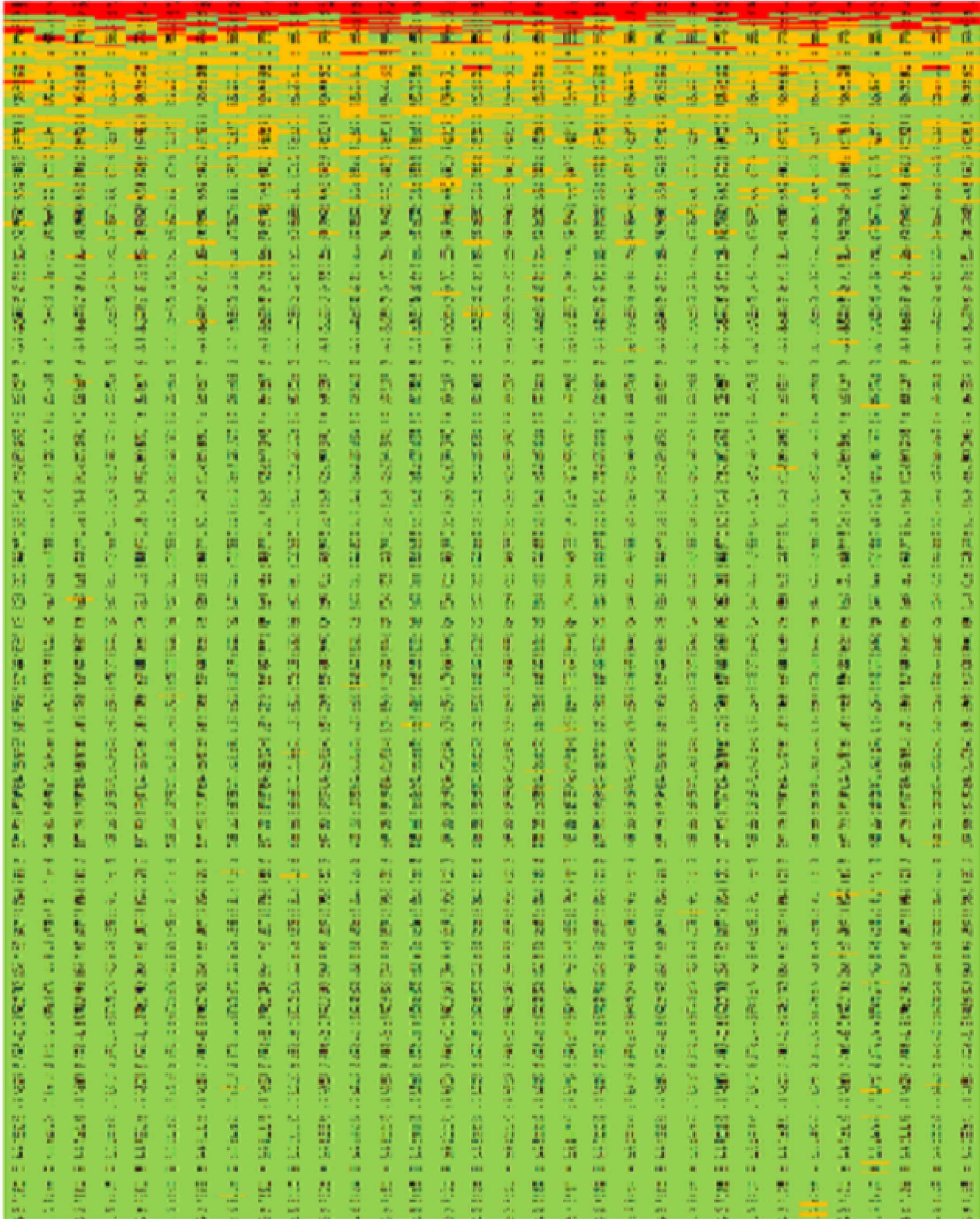


Figure B.11: Avalanche Matrix: Murmur2 with Bias DataSet

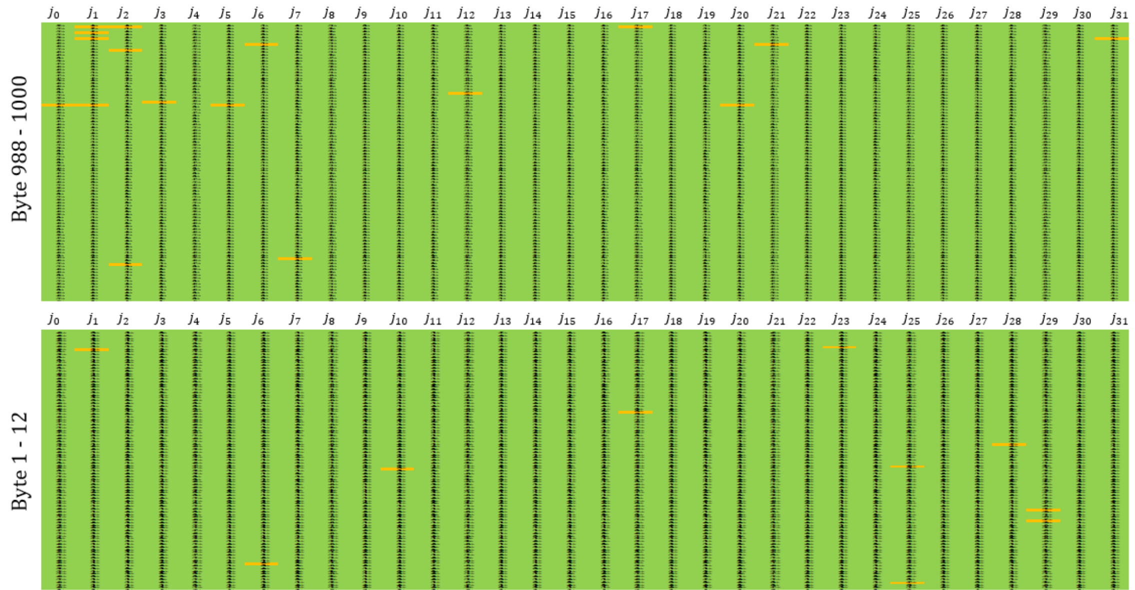


Figure B.12: Avalanche Matrix: Murmur2 with IP Addresses DataSet

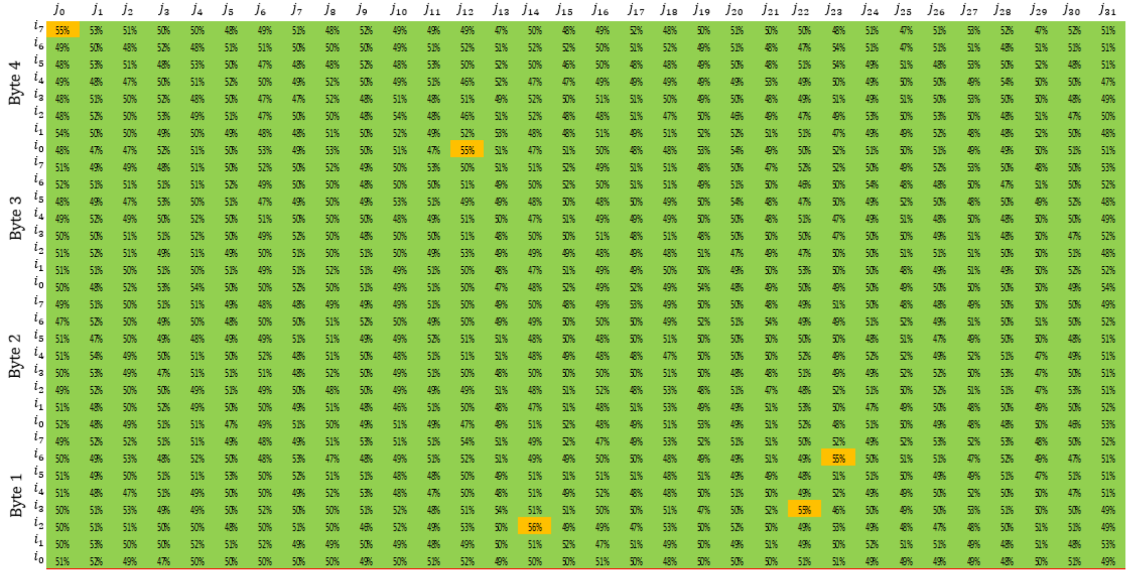


Figure B.13: Avalanche Matrix: DJBX33A with Baby Names DataSet

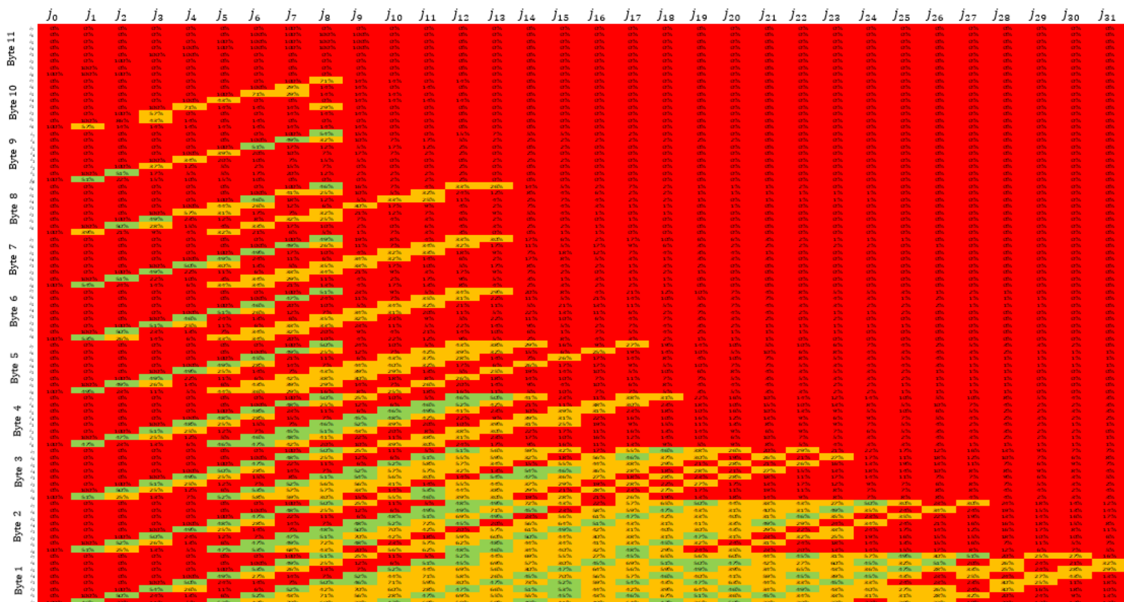


Figure B.14: Avalanche Matrix: DJBX33A with Common Words DataSet

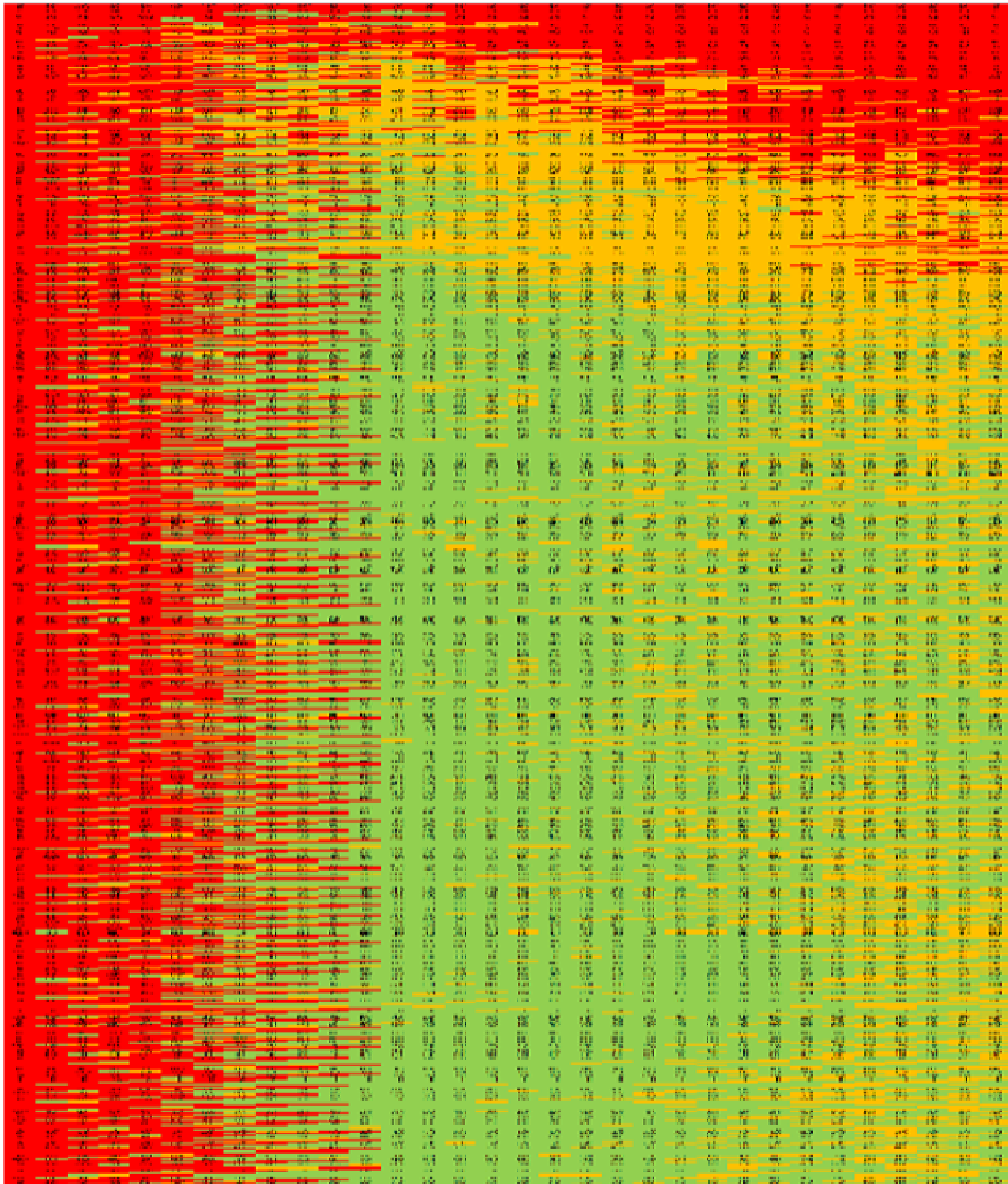


Figure B.15: Avalanche Matrix: DJBX33A with Bias DataSet

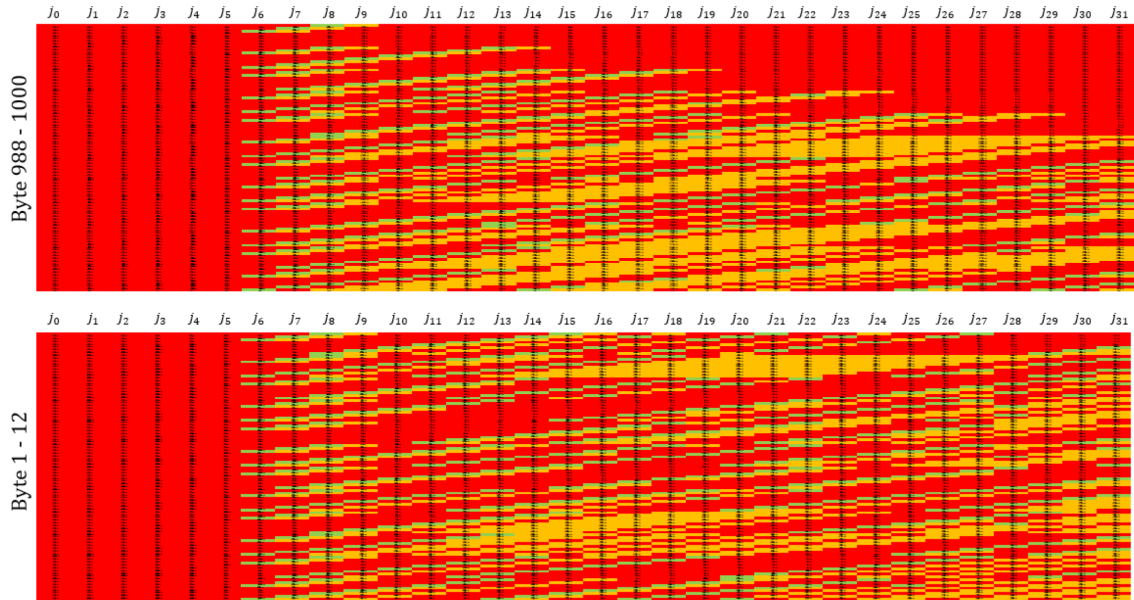
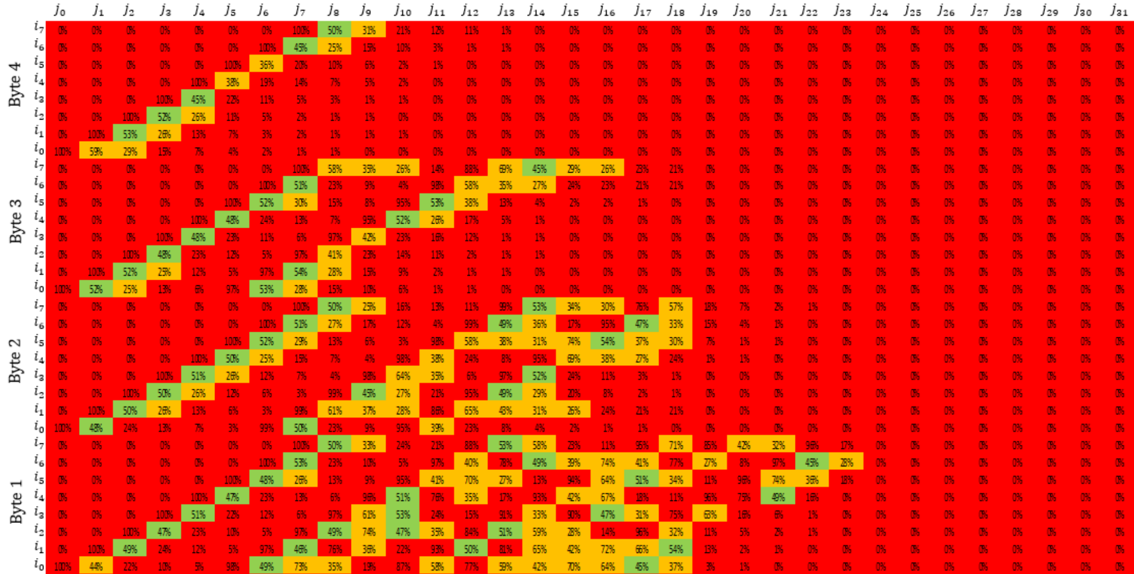


Figure B.16: Avalanche Matrix: DJBX33A with IP Addresses DataSet



Appendix C

Poisson Expectations vs Observed Results for 499 and 512 buckets

As shown on page 49, the results observed for the number of empty buckets and collisions when distributed among 500 buckets was roughly in line with what the Poisson Model had indicated we should expect.

The same is true for when using both 499 and 512 buckets, results are shown in Table C.1 and C.2 below. As before, we measure the results from the *Baby Names, Common Words* and *IP Addresses* input data sets only, as the *Bias* data set has been shown not to follow uniform distribution.

Table C.1: Summary of results for 499 buckets

Empty Buckets	Maximum	74
	Average	63
	Minimum	49
Collisions	Maximum	314
	Average	302
	Minimum	291
Longest Chain	Maximum	9
	Average	7.3
	Minimum	6

The Poisson Model (see details on page 35) would indicate that we should expect to see 67 empty buckets and 297 collisions when using 499 buckets.

Table C.2: Summary of results for 512 buckets

Empty Buckets	Maximum	83
	Average	69
	Minimum	54
Collisions	Maximum	313
	Average	298
	Minimum	290
Longest Chain	Maximum	8
	Average	6.8
	Minimum	6

The Poisson Model in this case would indicate that we should expect to see 73 empty buckets and 298 collisions when using 512 buckets.

Appendix D

DJBX33A Collision Resistance

As described in Section 7.3.2, DJBX33A exhibited some interesting patterns when tested for collision resistance. See table D.1 below as a reminder.

Table D.1: Distribution of DJBX33A for all possible inputs

Distribution of all possible 4-byte inputs		
		DJBX33A
2-byte	# Generated Outputs	8,671
	# Single Output	**66**
	# Collisions	8,605
3-byte	# Generated Outputs	286,366
	# Single Output	**66**
	# Collisions	286,300
4-byte	# Generated Outputs	9,450,301
	# "Empty Buckets"	4,285,516,995
	# Single Output	**66**
	# Collisions	9,450,235

To understand this pattern of just 66 distinct hash outputs regardless of input size, we consider the number of ways a particular output can be achieved. As we are looking at every possible permutation, let's sort them and look at the first three of the possible two-byte inputs:

$$(00000000\ 00000000)$$

$$(00000000\ 00000001)$$

$$(00000000\ 00000010)$$

DJBX33A starts from zero, multiplies by 33 and then adds the next byte (recall pseudocode on page 31). So the above three inputs are treated as follows (ignoring the adjustment required for 32-bit output for the sake of clarity and space):

$$(((0 * 33) + 00000000) * 33) + 00000000 = 0000000000000000$$

$$(((0 * 33) + 00000000) * 33) + 00000001 = 0000000000000001$$

$$(((0 * 33) + 00000000) * 33) + 00000010 = 0000000000000010$$

This is the only possible route to achieve these specific outputs, and the same can be said for each of the first 33 sorted inputs. However, once we go beyond the first 33, there becomes more than one way to achieve each output, leading to collisions. For example,

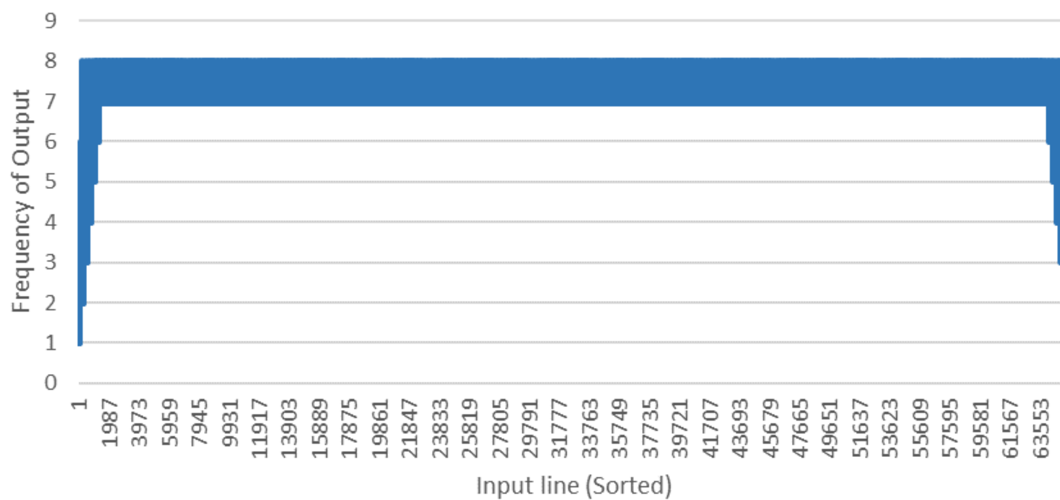
$$(((0 * 33) + 00000001) * 33) + 00000000 = 0000000000100001$$

$$(((0 * 33) + 00000000) * 33) + 00100001 = 0000000000100001$$

So only the first 33 inputs will always result in distinct outputs, and the same logic can be used for the final 33 inputs from our sorted list. Hence, in total, we will only ever see 66 distinct outputs, regardless of input size.

When examining the pattern of collisions for all two-byte inputs, we saw (as above) that the first 33 inputs each resulted in a distinct output. The next 33 inputs then each saw 2 colliding outputs. The following 33 inputs had 3 colliding outputs and so on, up to the maximum of 8 matching outputs. After 33 inputs hitting the maximum chain length of 8, we then dipped back to a chain length of 2 for the next 33 inputs, followed by 3 colliding outputs and so on again up to 8. This pattern is mirrored at the other end of the sorted input list, resulting in the pattern seen in Figure D.1

Figure D.1: Pattern of Chain Lengths for DJBX33A 2-byte inputs



Bibliography

[1] D. Wong, *Real World Cryptography.* Manning Online, ch. 2.2 Security properties of a hash function.

[2] B. Mulvey, "The Pluto Scarab," https://papa.bretmulvey.com/post/124027987928/hash-functions.

[3] J. Katz and Y. Lindell, *Introduction to Modern Cryptography.* CRC Press, 2014, ch. 5.2: The Merkle-Damgard Transform.

[4] R. Rivest, "The MD5 message-digest algorithm," https://www.ietf.org/rfc/rfc1321.txt.

[5] D. Eastlake and T. Hansen, "US secure hash algorithms (SHA and SHA-based HMAC and HKDF)," https://datatracker.ietf.org/doc/rfc6234/.

[6] R. Awati, "Password Salting," https://www.techtarget.com/searchsecurity/definition/salt.

[7] M. Mitzenmacher and E. Upfal, *Probability and Computing.* Cambridge University Press, 2017, ch. 5.1 Example: The Birthday Paradox.

[8] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer, "The sponge and duplex constructions," https://keccak.team/sponge_duplex.html.

[9] D. Wong, *Real World Cryptography.* Manning Online, ch. 2.5.2 The SHA-3 hash function.

[10] J. Guo, P. Karpman, I. Nikoli, L. Wang, and S. Wu, "Analysis of BLAKE2," https://eprint.iacr.org/2013/467.pdf.

[11] A. Hartikainen, T. Toivanen, and H. Kiljunen, "Whirlpool hashing function," https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=eae4ca3441b83c14e17c6866d0214b623195bdfd.

[12] J. Katz and Y. Lindell, *Introduction to Modern Cryptography.* CRC Press, 2014, ch. 5.5: The Random Oracle Model.

[13] T. Pornin, "What is the random oracle model and why is it controversial?" https://crypto.stackexchange.com/questions/879/what-is-the-random-oracle-model-and-why-is-it-controversial.

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

[15] 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.

[16] K. A. G. Classroom, "Lagrange multipliers, introduction," https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/constrained-optimization/a/lagrange-multipliers-single-constraint.

[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.

[18] 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.

[19] 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/.

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

[21] C. S. Office, "Top baby names: Boys and girls," https://visual.cso.ie/?body=entity/babynames.

[22] M. Languages, "1000 most common english words," https://1000mostcommonwords.com/1000-most-common-english-words/.

[23] 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.

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

[25] ——, "MurmurHash3.cpp," https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[26] "Which hashing algorithm is best for uniqueness and speed?" https://hackerthink.com/solutions/which-hashing-algorithm-is-best-for-uniqueness-and-speed/.

[27] M. Mitzenmacher and E. Upfal, *Probability and Computing*. Cambridge University Press, 2017, ch. 5.2 Balls Into Bins.

[28] ——, *Probability and Computing*. Cambridge University Press, 2017, ch. 5.3 The Poisson Distribution.

[29] T. W. Anderson and D. Darling, "Anderson-Darling Test," https://www.itl.nist.gov/div898/handbook/eda/section3/eda35e.htm.

[30] C. E. Bonferroni, "Bonferroni Correction," https://www.statisticssolutions.com/bonferroni-correction/.

[31] Y. Benjamini and Y. Hochberg, "Benjamini-Hochberg Procedure," https://www.statology.org/benjamini-hochberg-procedure/.

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

[33] 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.

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

[35] 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.

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

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

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

[39] Y. Romailler, "THE DEFINITIVE GUIDE TO "MODULO BIAS AND HOW TO AVOID IT"!" https://research.kudelskisecurity.com/2020/07/28/the-definitive-guide-to-modulo-bias-and-how-to-avoid-it/.