FASH64

Douglas Crockford and Aashish Sheshadri

November 6, 2017

Abstract

`Fash64` is an efficient hashing function. It processes 64 bit blocks at a time to produce a 64 bit result. It can be used for implementing data structures (hash tables) and checksums. `Fash64` relies on multiplication. In a single instruction, 64 bit multiply can do up to 64 left shifts and 64 additions. On most CPUs, the product of multiplication can be 128 bits, divided over two registers. `Fash64` uses the high 64 bits of the product as a 64 bit right shift that can quickly be fed back into the hash. Without that massive right shift, multiply tends to lose information that spills out the left, which would make it unsuitable for hashing. But we get good feedback from that high part, yielding good hashes.

1 Introduction

Hash functions are plentiful and well studied. They often seem to be a work of art with magic numbers studded over a carefully ordered sequence of instructions. They are hardly simple and almost always highly sensitive to any change. An excellent survey[1] by Bob Jenkins[4] details the many available hashing methodologies which continues to be a growing set.

This paper presents, `FASH64`, a practical hash function that is simple, fast and reliable. `FASH64` is an efficient hashing function. It crunches 64 bits at a time to produce a 64 bit result. It can be used for implementing data structures (hash tables) and checksums. The novelty lies in employing 64 bit multiplication and using the full 128 bit product to enable excellent dispersion. A hashing algorithm that also uses multiplication and is quite well studied is the FNV[6] Hash. FNV however has been shown to be statistically weak in work by Colin Pesyna[7]. `FASH64` differs in a few of respects, 1) It process 64 bits at a time instead of the 8, 2) It is not sensitive to the choice of the prime, and most importantly, 3) It does not rely on multiplication alone. We do include the prime used by FNV-1a 64 in our tests.

Section 2 introduces `FASH64` and details its internals. Section 3 presents a thorough analysis of the hashing ability of `FASH64`. Through rigorous statistical tests `FASH64` is shown to be sound for practical applications and may be a viable candidate for secure hashing. However, secure hashing is out of the scope of this paper.

[1]http://burtleburtle.net/bob/hash/doobs.html

2 Fash64

Fash64 relies on multiplication. In a single instruction, 64 bit multiply can do up to 64 left shifts and 64 additions. On most CPUs, the product of multiplication can be 128 bits, divided over two registers. Fash64 uses the high 64 bits of the product as a 64 bit right shift that can quickly be fed back into the hash. Without that massive right shift, multiply tends to lose information that spills out the left, which would make it unsuitable for hashing. But we get good feedback from that high part, yielding good hashes. The workhorse function of FASH64 in pseudo code is presented in Listing 1.

Listing 1: FASH64 Pseudo Code.

```
prime := 11111111111111111027 # a large prime number
result := 8888888888888888881 # initial value
accumulator := 3333333333333333271 # initial value

def fash64_word(word: uint64) {

# Takes a 64 bit word, and scramble it into the hash.

    high: uint64 # The high part of the product
    low: uint64 # The low part of the product

# Mix the word with the current state of the hash
# and multiply it with the big prime.

    high; low := (result xor word) * prime

# Add the high part to the accumulator.
# This is to defend against result equaling
# the word, which would cause loss of all
# memory of the previously hashed stuff.

    accumulator += high

# Mix the low part with the sum.

    result := low xor accumulator
}
```

Most CPUs know how to do a multiply that produces a 128 bit product, but most programming languages do not, tossing away the valuable high bits. It is tragic that popular languages do not allow a statement like `high; low := (result xor word) * prime` that deposits the product of the multiplication into the `high` and `low` variables.

The `accumulator` variable deals with the possibility of `result xor word` producing *zero*. If we used `result := low xor high` then `result` can become *zero* when `result` *equals* `word`, which loses the influence of everything that was hashed up to this point. We mitigate this with `accumulator += high` and `result := low xor accumulator`. The `accumulator` variable retains the influence of the earlier material, so the hash will still be good. This borrows an idea from Fletcher's Checksum[3]. The likelihood that a `word` will match the current `result` and cause a reset is 1 in 2^{64}. The `accumulator` makes a reset even less likely.

3 Experimental Analysis and Results

In this section we present a suite of statistical tests that enable strict evaluation of `FASH64`. The mixing step in one of the main operations of `FASH64` (See Listing 1). The only parameter used is the prime in the 64 bit multiplication operation. In order to surface sensitivity to the choice of the prime we introduce two additional variations of `FASH64`, the `FASH64_WEAK_PRIME` and `FASH64_ALT_PRIME` that differ from `FASH64` in the prime number used.

`FASH64_WEAK_PRIME` uses the prime 1099511628211. The inclusion of this prime is attributed to its critical improvement of FNV-1a[6]. It is not capable of mixing 64 bits well; it is not 64 bit in length, has only 17% of bits set and is unevenly distributed. One possible reason for its stellar performance with FNV-1a may be attributed to limiting mixing to 8 bit chunks. In contrast, `FASH64` uses `11111111111111111027` as its prime; the prime has 53% of its bits set, with excellent dispersion of the set bits. Finally, `FASH64_ALT_PRIME` uses `9999999999999999961` as its prime. Similar to choosing a prime for `FASH64`, this prime was chosen to be 64 bit in length and about 50% of the bits set to 1. This prime however exhibits a slightly uneven distribution of set bits, biased to the lower end.

Additional primes satisfying similar criteria were found to perform almost identically and hence were omitted from presentation to save space. A few of the other comparable primes tested include,

- `6666666666666666619`
- `7777777777777777687`
- `5555555555555555533`
- `4444444444444444409`
- `2222222222222222177`

The focus of this paper however is not to find the best possible prime for `FASH64`, however we find sensitivity to be minimal when the chosen prime met the criteria of being at least 64 bit in length and had an even distribution of 1's and 0's.

Section 3.1 introduces the dataset used for all of the following experiments. Section 3.2.1 measures avalanching ability of the proposed hash function. Section 3.3 measures the ability of `FASH64` to build hash tables. Finally, Section 3.4 measures bit correlation across hashes computed by `FASH64`.

3.1 Data

This paper measures statistical properties of hashes computed on both real world data and synthetic data. Real world data enables evaluation on biased distributions found in daily hashing requirements. In contrast, synthetic data enables measurement on uniformly distributed bytes. Each of the presented

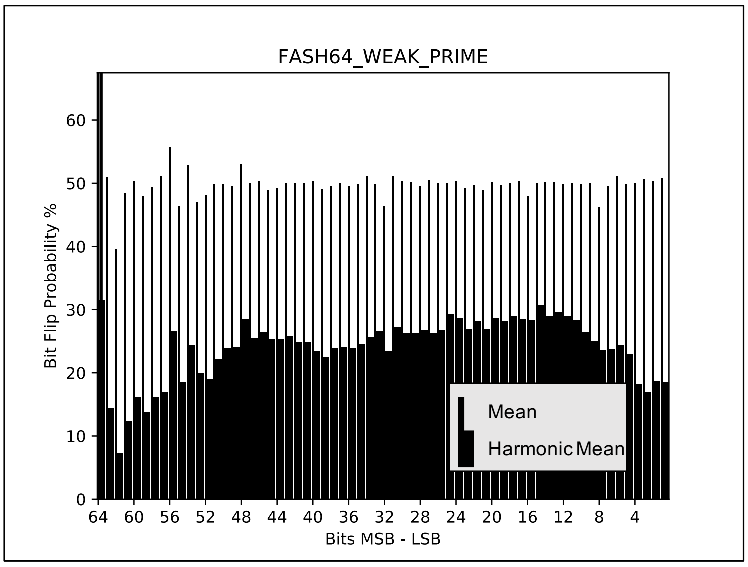


Figure 1: Mean and harmonic mean of bit set probabilities of 64 bit hashes computed using `FASH64_WEAK_PRIME` on perturbed passwords with unit Hamming distance.

experiments make fine modifications to the data to study various statistical properties.

3.1.1 Real World Data

To enable real world evaluation, we used the top 1M passwords from the 10M password list[2] as test vectors. The dataset is a compilation of compromised passwords from recent public dumps.

3.1.2 Synthetic Data

A typical use of hash functions is to create message digests. Messages of length 512 bytes were generated from a uniform random distribution to enable measurement of performance on this criterion.

3.2 Measuring Avalanche

Avalanche in the context of hashing can be informally defined as a phenomenon that causes a large change in the output hash for a small change in the input data. This behavior is a necessity for practical hash functions and required for a cryptographic hash function. To test the proposed hash function we adopt the Strict Avalanche Criterion(SAC) introduced by Webster et al.[8].

Fulfilling the SAC criterion requires that given any input X, and the corresponding hash Y; complementing the ith bit $\forall i$ in X should complement each bit in Y with a probability of 0.5.

[2]https://github.com/danielmiessler/SecLists/tree/master/Passwords

Since we expect FASH64 to function as a reliable practical hash, we chose as input vectors, common passwords used on the web (Section 3.1.1). Section 3.2.1 describes the experiment and results. Section 3.2.2 describes an experiment which extends the SAC to uncover possible funnels[3] of up to 3 bits. Funnels result in hash collisions when, $n > m$ and n bits of X only affect m bits of Y.

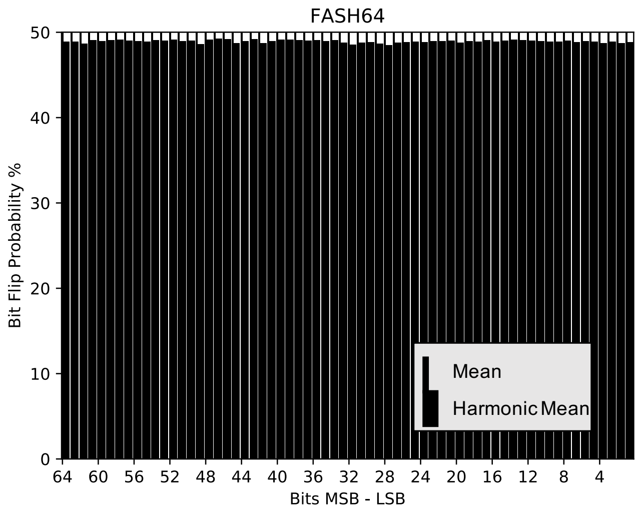


Figure 2: Mean and harmonic mean of bit set probabilities of 64 bit hashes computed using `FASH64` on perturbed passwords with unit Hamming distance.

3.2.1 Measuring Avalanche on Passwords

The experiment was set up to measure FASH64's adherence to the SAC. Web passwords were chosen as test vectors (Section 3.1.1). Given a password X_i, let X_{ij} be the result of complementing the j^{th} bit. Vectors with a hamming distance of 1 is the set of all such X_{ij}. Given the set of test vectors, hashes were computed using `FASH64`, `FASH64_WEAK_PRIME` and `FASH64_ALT_PRIME`.

Let Y_i be the hash computed on X_i and similarly Y_{ij} be the hash computed for X_{ij}. To check adherence to the SAC, the probability of a bit being set was measured across all computed hashes. We expect the probabilities to measure close to 50% if the SAC is to be satisfied. Figure 2 plots mean and the harmonic mean of measured bit probabilities on computed `FASH64` hashes on the 1M passwords. Harmonic mean is reported to uncover outliers. Similarly, Figure 1 and Figure 3 plots measured bit probabilities when using `FASH64_WEAK_PRIME` and `FASH64_ALT_PRIME` as the hash function respectively.

Figure 2 clearly indicates `FASH64`'s performance to be compliant with the SAC. Both the mean and the harmonic measured close to if not exactly 50% for each bit with no measurable bias towards any bit. Alternatively, using `FASH64_ALT_PRIME` introduced some variation on the harmonic mean measurement (See Figure 3), but the mean bit probabilities measured 50%. Upon further

[3]http://burtleburtle.net/bob/hash/funnels.html

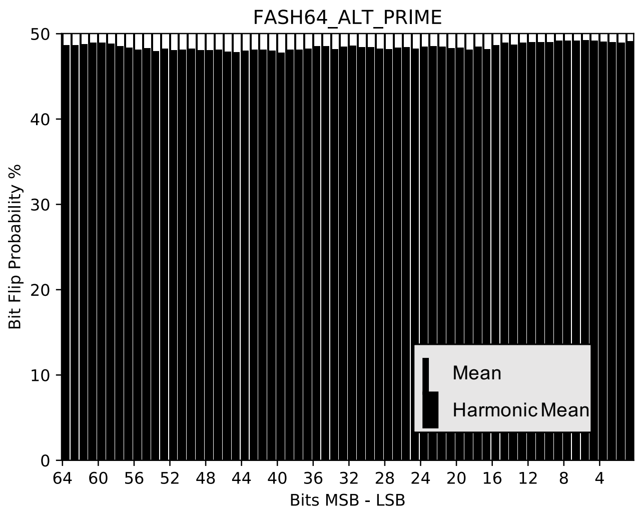
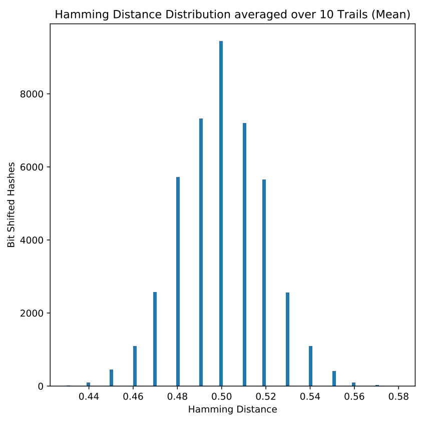
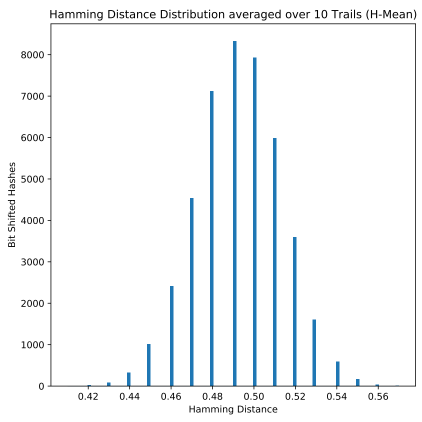


Figure 3: Mean and harmonic mean of bit set probabilities of 64 bit hashes computed using `FASH64_ALT_PRIME` on perturbed passwords with unit Hamming distance.



(a) Mean over 10 trials.



(b) Harmonic mean over 10 trials.

Figure 4: Histogram of computed hamming distance (normalized) measures between the original message hash and hashes of perturbed messages with a maximum hamming distance of 3. Figure 4a plots the mean hamming distance measure over 10 trials and Figure 4b plots the harmonic mean over 10 trials. Hashes were computed using `FASH64`.

investigation there were 10 instances where a few bits of the output hash remained invariant to input perturbations. These instances caused the harmonic mean to be undefined. To overcome this, minimum bit probability was set to 0.01, to enable computation of the harmonic mean while ensuring penalization. Interestingly, the 10 instances were the only single byte passwords in the dataset. `FASH64` did not show this behavior and there were no instances where output hash bits remained invariant to input perturbations. This does not however imply that there aren't instances that can causes the output hash bits to be invariant. We can though be confident that this occurrence has a pretty low probability.

As can be seen in Figure 1, `FASH64_WEAK_PRIME` showed significant bias in measured bit probabilities. Alarmingly, stagnation of a few lower order and higher order bits was evident on a lot of instances. This greatly impacts the ability to truncate the hash. While the measured mean bit probabilities seem to acceptable for most bits, stagnation of bits 64^{th} (being disproportionately high) and 62^{nd} (being disproportionately low) makes these bits unsuitable. The behavior can be attributed to poor mixing. Specifically, the multiplication with the prime is a step critical to enable good mixing. A poor prime results in poor mixing and resultant bias. The harmonic mean measures quite dismally across all bits, indicating the presence of several instances where output hash bits were invariant to input perturbations.

For the rest of the paper, owing to similar results observed between `FASH64` and `FASH64_ALT_PRIME`, we only report results for `FASH64`. The better performance of `FASH64` over `FASH64_ALT_PRIME` in this experiment may be attributed to the fact that the bits are more evenly distributed within the prime. We however forgo studying primes in this paper and leave it to future work, especially since there was little to no observed benefit in the rest of the experiments. We will continue to compare `FASH64` with `FASH64_WEAK_PRIME` to show sensitivity to effective mixing while computing a hash.

3.2.2 Measuring Avalanche on Messages

The experiment was designed to uncover possible funnels causing hash collisions. Instead of limiting input perturbations to a hamming distance of 1, we included all possible test vectors up to a hamming distance of 3. Like the SAC criterion, we required that the measured bit probabilities on computed hashes tend to 50%.

Synthetic data was generated as messages (See Section 3.1.2), totaling to a count of 10 messages to average over results. Synthetic data was used since the set of test vectors with a hamming distance of up to 3 is quite large. Picking just 10 real world passwords will be biased, requiring the need to average over several samples. The use of uniformly distributed random data makes this less of a concern. Synthetic data also enabled control over the size of the message which was set at 512 bytes. Furthermore, using this data enabled a complementary view to that described in the previous experiment.

For each 512 byte message, the middle 8 bytes (64 bits) were perturbed to enumerate all possible messages with a maximum hamming distance of 3. Using the combinatorial notation ${}^{n}C_{k}$ representing *n choose k*, we have ${}^{64}C_{1}$ messages with one of the 64 bits perturbed resulting is set of messages with a hamming distance of 1, ${}^{64}C_{2}$ messages with any two of the 64 bits perturbed resulting in

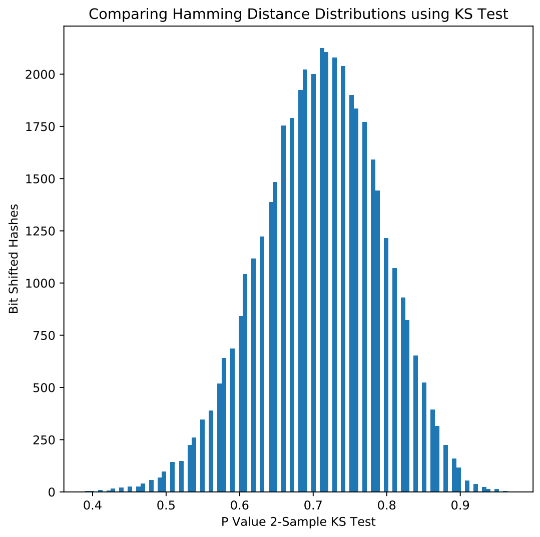


Figure 5: Histogram of p-values computed using the two sample Kolmogorov-Smirnov test to measure statistical similarity of two distributions. Each p-value corresponds to a similarity measure of the distribution presented in Figure 4a and the distribution of hamming distances computed between the hash of a perturbed message with the rest of the messages using `FASH_64`.

a set of messages with hamming distance of 2 and, finally, ${}^{64}C_3$ messages with any three of the 64 bits perturbed resulting in a set of messages with a hamming distance of 3 from the original message. Hence, given a message, the perturbed messages enumerated to $43,744$,

We report results from a perspective alternative to that described in Section 3.2.1. We measured the hamming distance of hash pairs drawn from all the hashes computed for the perturbed messages. While we lose bit level information on a computed hamming distance measure, this enables a closer look at the space of all pairwise comparisons. Note that hamming distances reported have been scaled to lie in the range $[0, 1]$, i.e., a hamming distance of 0.5 translates to 50% of the bits being different.

To illustrate the experiment, let Y be the hash computed on a message, and Y_i be the hash computed on the i^{th} perturbed message; i enumerates all $43,744$ perturbed messages. We have ${}^{43745}C_2$ (this includes Y) possible hash pairs and a hamming distance measure for each pair. This can be represented as a 2D hamming distance matrix H of dimension 43745 x 43745. Note that H is a symmetric matrix since hamming distance is commutative. The diagonal of this matrix represents the hamming distance of a hash with itself and hence will be zero. Let (0,0) in H represent the hamming distance between Y and Y, then (0,1) will encode the the hamming distance between Y and Y_1 and so on. Thus, this row vector H_0 compares Y with all the hashes computed on perturbed messages. Similarly, the second row, H_1, is a vector encoding hamming distance between Y_1 and all the hashes and so on.

This being a large space to visualize, we present results to capture the distribution of hamming distances in a statistically equivalent manner. We first present in Figure 4a the distribution of hamming distances computed between Y and each Y_i (encoded in H_0) averaged across 10 messages. Similarly, in Figure 4b, we present the harmonic mean across the 10 messages as opposed to a simple average. There exists a distribution in each row vector of H. To circumvent having to visualize each distribution, Figure 5 plots the p-values computed using the Two-Sample Kolmogorov-Smirnov test[5] comparing the distribution represented by the i^{th} row vector in H with the H_0 distribution presented in Figure 4a and Figure 4b. The mean p-value was measured as 0.71 with a standard deviation of 0.08. A p-value greater that 0.05 suffices to show that the distributions are statistically similar. We thus restrict our analysis to H_0 and have the same results be applicable to distributions represented by the rest of the row vectors in H.

The hamming distance measures plotted in Figure 4a have a mean of 0.50 and a standard deviation of 0.01. This shows that the message hash Y and the perturbed hashes $\{Y_1, .., Y_i, ..., Y_{43744}\}$ are highly uncorrelated and uniformly random. Figure 4b which plots the harmonic mean across the 10 message samples measures similarly with a mean of 0.49 and standard deviation of 0.02, thus ensuring the absence of outliers.

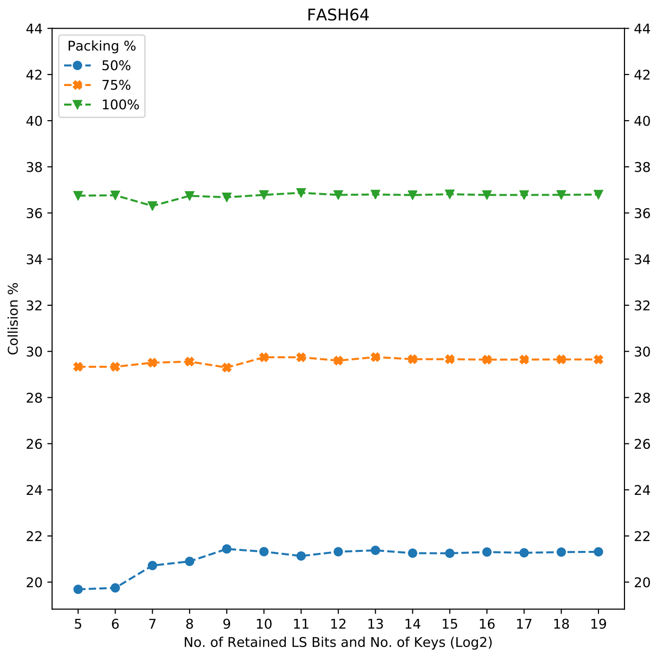


Figure 6: Measuring collision when using `FASH_64` as the hash function to build a hash table with fixed 2^n capacity.

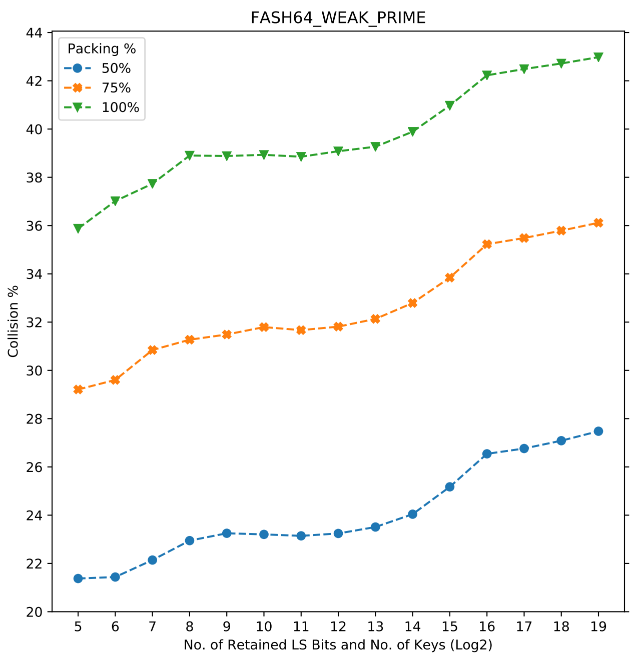


Figure 7: Measuring collision when using `FASH_64_WEAK_PRIME` as the hash function to build a hash table with fixed 2^n capacity.

3.3 Measuring Collision

One of the use cases of a practical hash function is to enable building efficient hash tables. This section presents a real world evaluation by measuring and reporting statistics when building a hash table using passwords as keys. The metric we focus on is collision rate, i.e, percentage of keys that are hashed to the same value.

The experiment was set up to measure collision rate on a key space that was 50%, 75% and 100% full. We chose key spaces as powers of 2 from 2^5 through 2^{19}; the upper limit was dictated by the number of available keys (1M). Section 3.1.1 details the nature and source of the data. The hash size was restricted to n bits enumerating 2^n buckets. The motivation was to have just enough buckets to hash keys with out collision. A *perfect hash function*(PHF)[2] can hash keys with out collision given greater than or equal to 2^n buckets. However, given exactly 2^n buckets only a *minimal perfect hash function*(MPHF) will be able to hash without collision. FASH is neither a PHF or a MPHF and hence exhibits collision under the test conditions. The objective of this experiment is to measure the collision rate and study its dependence on the size of the key space and available buckets. Note that FASH and other hashing algorithms that can uniformly distribute keys can be converted to a practical MPHF[1]. Figure 6 reports collision rates using `FASH64` for each key space. Similarly, Figure 7 reports collision rates using `FASH64_WEAK_PRIME`. We also measured impact on collision rate with a fixed key space of 2^{19} but increasing buckets from 2^{19} through 2^{24}. Figure 8 plots these results when using `FASH64`. All reported

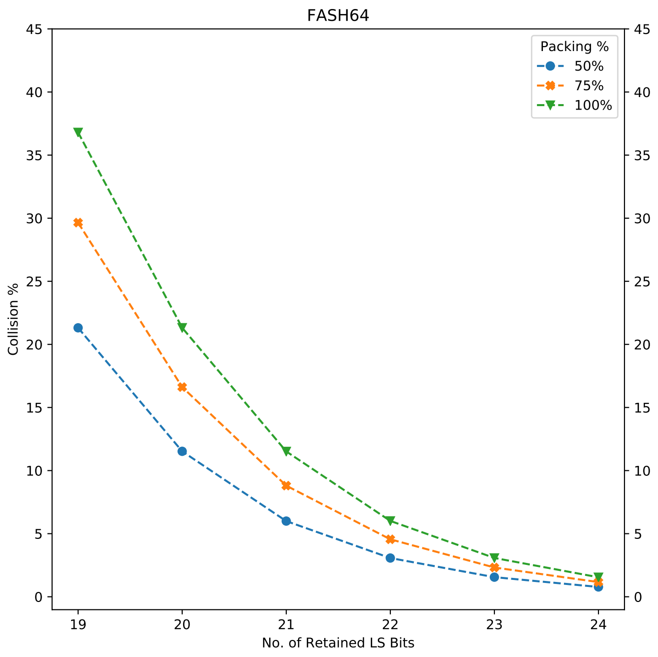


Figure 8: Measuring collision when using `FASH_64` as the hash function to build a hash table with increasing capacity but fixed key space of 2^{19}.

results were averaged over 100 trials. Key space for each trial was populated by a random sample drawn uniformly from the 1M passwords.

Figure 6 shows that as the key space doubles along with the buckets, the collision rate of `FASH64` does not change significantly and is quite stable. This behavior suggests that distribution of the keys is a process limited by the number of buckets available. This is evident in the collision rate falling by nearly 50% when the key space is halved or when the number of buckets is doubled (See Figure 8). Doubling the buckets has the impact of resolving 50% of the previously colliding keys. This behavior suggests a uniform distribution of the keys in the hash space.

Without good mixing enabled by a strong prime we see that the collision rate is higher and not quite stable with increasing number keys and proportional buckets as shown in Figure 7. Collision rate increases with number of keys even with a proportional increase in the number of buckets because of inherent bias in key distribution.

3.4 Measuring Correlation

On average we expect there to be no correlation among any bit groups in a hash. To measure correlation, we divided each computed hash into four equal parts of 16-bit length. The higher order 32 bits were split as the higher order 16 bits (*high*), the rest of the 16 bits (*mid high*). The lower order 32 bits were split as the lower order 16 bits *low*, the rest of the 16 bits *mid low*. Each of the bit groups were considered as 16-bit unsigned integers. We then plotted them

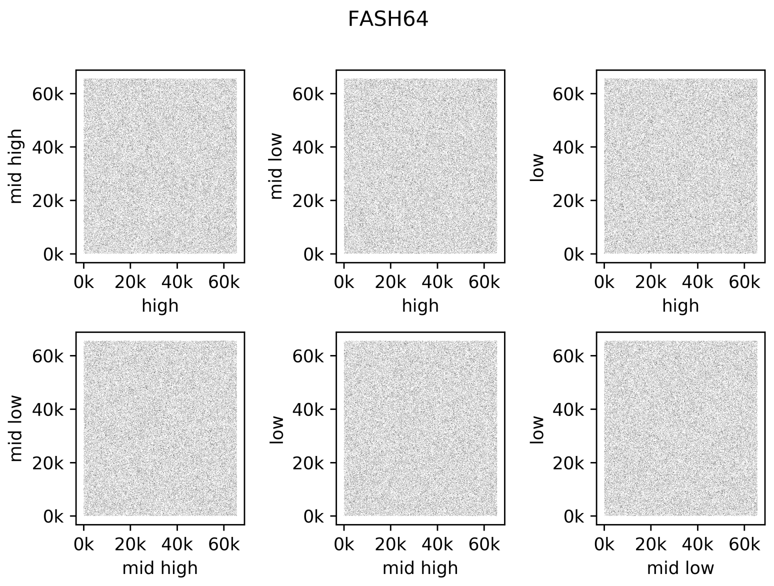


Figure 9: A scatter plot comparing four sections of 1M 64-bit FASH64 hashes(*high*, *mid high*, *mid low*, *low*) for correlation to uncover bias. There is no discernible structure in any of the figures which is a strong indication of absent bias or correlation between any bit groups.

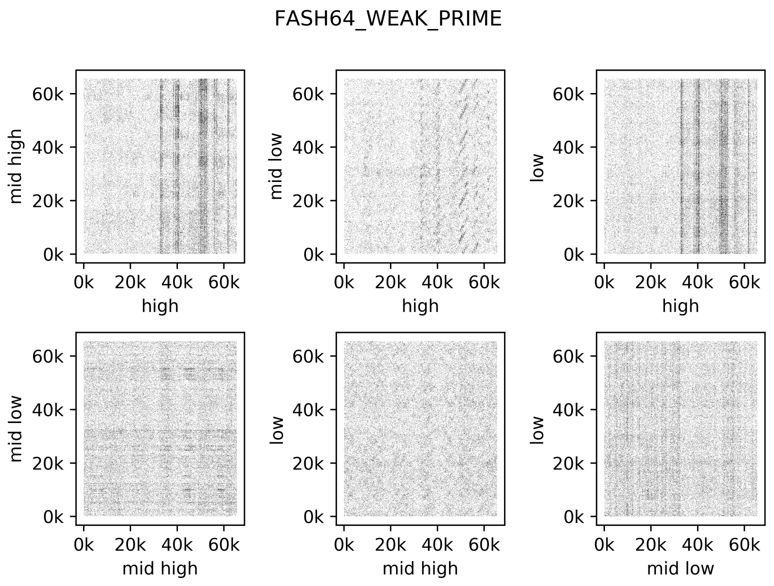


Figure 10: A scatter plot comparing four sections of 1M 64-bit FASH64_WEAK_PRIME hashes(*high*, *mid high*, *mid low*, *low*) for correlation to uncover bias. Note the horizontal and vertical banding indicative of bias towards a bit group.

pairwise as (x, y) coordinates on a 2D scatter plot; one such plot was the *high* as the y-coordinate and *low* as the x-coordinate. Such a representation can uncover correlation, usually as banding. Hashes were computed on the 1M passwords (Section 3.1.1) using `FASH64` and `FASH64_WEAK_PRIME`. Figure 9 presents plots for all pairs on hashes computed using `FASH64`. Similarly, Figure 10, presents plots for hashes computed using `FASH64_WEAK_PRIME`.

As expected of a good hash function, hashes computed using `FASH_64` show no correlation between any of the possible pairs of bit groups. The scatter plot only shows a random 100,000 subset, since including all the 1M hashes resulted in saturation of every pixel. The absence of any structure on any of the plots is indicative of an absent bias. This further validates the ability of `FASH64` to uniformly distribute over the 64-bit hash space. As expected, using `FASH64_WEAK_PRIME` as the hash function exhibits bias across multiple bit groups. The bias manifests as banding in the scatter plot. Vertical banding is evident showing bias towards the *high* bit group along with some horizontal banding showing bias towards *mid low* bit group. This again stresses the importance of the mixing step in FASH64.

4 Discussion and Conclusion

We introduced `FASH64`, a new 64 bit hashing algorithm that crunches 64 bits at a time. In doing so, while most hashing schemes work one byte at a time, `FASH64` enables a 8-fold speed up. This is apart from the fact that each 64 bit block is processed in just nine instructions; three to load, four to compute and two to update saved state. The simple design of the algorithm, devoid of magic numbers and a multitude of finely devised instructions, enables stability over changing parameters. More importantly, it enables easy comprehension and reasoning.

In Section 3.2 we showed both on real world data and uniformly distributed synthetic data that small perturbations to the input has a avalanching effect on the computed hash. Apart from demonstrating adherence to the SAC, we also showed all the computed hashes were uniformly distributed. In Section 3.3 we showed `FASH64` to be dependable and efficient in the context of building hash tables. Through experimentation over varying hash table sizes and key spaces we showed expected collision rates and provided intuition on handling collision. Finally, in Section 3.4 we showed no two bit groups to be correlated on average over a million hashes. Through these experiments we have thus comprehensively showed `FASH64` to be statistically sound. We therefore strongly assert `FASH64` to be a fast, reliable and efficient candidate for practical hashing.

References

[1] Botelho, F. C., and Ziviani, N. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management* (2007), ACM, pp. 653–662.

[2] Czech, Z. J., Havas, G., and Majewski, B. S. Perfect hashing. *Theoretical Computer Science 182*, 1-2 (1997), 1–143.

[3] Fletcher, J. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications 30*, 1 (1982), 247–252.

[4] Jenkins, B. Algorithm alley. `http://www.drdobbs.com/database/algorithm-alley/184410284`, 1997.

[5] Lopes, R. H., Reid, I., and Hobson, P. R. The two-dimensional kolmogorov-smirnov test.

[6] Noll, L. C. Fowler/noll/vo (fnv) hash.

[7] Pesyna, C. Choosing a good hash function. `https://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3/`, Sep 2013.

[8] Webster, A., and Tavares, S. E. On the design of s-boxes. In *Conference on the Theory and Application of Cryptographic Techniques* (1985), Springer, pp. 523–534.