Fast keyed hash/pseudo-random function using SIMD multiply and permute

J. Alakuijala, B. Cox, and J. Wassenberg

Google Research

November 20, 2017

Abstract

HighwayHash is a new pseudo-random function based on AVX2 multiply and permute instructions for thorough and fast hashing. It is 3.8 times as fast as SipHash for 1 KB inputs. An open-source implementation is available under a permissive license. We discuss design choices and provide statistical analysis, speed measurements and preliminary cryptanalysis. Assuming it withstands further analysis, strengthened variants may also substantially accelerate file checksums and stream ciphers.

1 Introduction

Hash functions are widely used for message authentication, efficient searching and 'random' decisions. When attackers are able to find collisions (multiple inputs with the same hash result), they can mount denial of service attacks or disturb applications expecting uniformly distributed inputs. So-called 'keyed-hash' functions prevent this by using a secret key to ensure unpredictable outputs. These functions are constructed such that an attacker who controls the inputs still cannot deduce the key, nor predict future outputs. The authors of SipHash refer to these as 'strong pseudo-random functions' [1]. However, existing approaches are too slow for large-scale use. In this paper, we introduce two alternatives that are about 2 and 4 times as fast as SipHash.

We are mainly interested in generating random numbers and message authentication, for which 64-bit hashes are sufficient. These are not 'collision-resistant' because adversaries willing to spend considerable CPU time could find a collision after hashing about $\sqrt{\frac{\pi}{2}2^{64}}$ inputs. However, small hashes decrease transmission overhead and are suitable for authenticating short-lived messages such as network/RPC packets. If needed, our approach can generate up to 256 hash bits at no extra cost.

Section 2 briefly discusses existing hash functions and their strength/speed tradeoff. Section 3 describes SipTreeHash, a j-lanes extension of SipHash that is twice as fast on large inputs. However, the SipHash construction is relatively slow because it relies on rotate instructions not available in SIMD instruction sets. Section 4 introduces HighwayHash, a novel hash function that takes advantage of SIMD permute instructions for fast and thorough mixing. Measurements in Section 5 indicate SipTreeHash is twice as fast as SipHash for large inputs, and HighwayHash almost four times as fast. Section 6 describes our test suite and shows that HighwayHash resists common attack techniques from the literature. Further study may require new cryptanalysis techniques.

2 Related Work

Current cryptographic hashes require at least 2-3 CPU cycles per byte [2], which is about an order of magnitude slower than fast hashes such as CityHash (0.23 c/b). Such a difference is unacceptable to practitioners, especially if they are unconcerned about security. SipHash [1] is a good compromise that has been studied since 2012 without any known weaknesses. Our implementation requires about 1.3 c/b. Although relatively inexpensive for a strong hash, this is still at least five times slower than fast hashes such as Murmur3 and CityHash. However, these are vulnerable to collision and key extraction attacks [3] and must not be exposed to untrusted inputs. Several approaches have subsequently been proposed for taking advantage of hardware-accelerated AES encryption [4, 5]. These include security proofs and are about twice as fast as SipHash. The recent CLHash [6] is even faster despite requiring 1064 byte keys. However, its mixing is insufficient to pass smhasher's avalanche test. A proposed fix adds an additional round of the ad-hoc Murmur mixing function, but this still fails our distribution test for zero-valued inputs (see Section 6.1). Note that CLHash was designed for speed and almost-universality, and is not intended to withstand attacks [7]. We believe that SipHash remains a good default choice for non-cryptographic applications because it offers (apparently) enough security at reasonable speeds. A version with only 1 update and 3 finalization rounds is 1.2 to 2 times as fast (see Section 5) while still passing smhasher, which makes it an interesting candidate for applications where security is less of a concern.

We develop two alternatives that further increase throughput while retaining the simplicity and thorough mixing of SipHash. We hope that these algorithms will replace unsafe hash functions and increase the robustness of applications without incurring excessive CPU cost.

3 SipTreeHash

Maximizing performance on modern CPUs usually requires the use of SIMD instructions. These apply the same operation (e.g. addition) to multiple 'lanes' (elements) of a vector. This works best for data-parallel problems. However, the SipHash dependency chain offers limited parallelism and cannot fully utilize the four AVX2 vector lanes. Our SIMD implementation was actually slower than the scalar version, presumably because of the lack of bit rotation instructions. We instead compute four independent hash results by logically partitioning input buffers into interleaved 64-bit pieces. For example, consider a 64 byte input interpreted as eight 64-bit words: $A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3$. These can be combined into four hash results with two updates, one with A_i and the other with B_i. We must then fold the four results into a single hash. XOR reduction is unsuitable because it cannot distinguish between permutations of the 64-bit words. Instead, we can just hash the results. This is known as a tree-hash construction and has been used to accelerate SHA-256 [8].

Our implementation [9] is available as open source software under the Apache 2 license. Being a straightforward extension of SipHash, this construction is likely to be secure. However, its hash results are of course different, so this cannot be used as a drop-in replacement. We suggest HighwayHash be considered instead because it is much faster, especially for smaller inputs.

4 HighwayHash

Tree hash constructions appear to be a good way to utilize SIMD instructions. However, the SipHash add-rotate-XOR construction is not ideally suited for current instruction sets. As previously mentioned, bit rotations must be implemented by ORing together the result of left and right shifts. Although rotations of multiples of 8 bits can be implemented with very fast byte permute instructions, this would weaken SipHash to an unacceptable degree [1]. The individual add and XOR instructions also only achieve a weak mixing effect. By contrast, AVX2 includes 32×32 bit multiplication instructions that mix their operands much more thoroughly. Although their latency is higher than non-SIMD multiplies (5 vs 3 cycles [10]), we believe the increase in mixing efficiency vs. add/XOR is still worthwhile. The 64-bit Intel architecture also provides 16 SIMD registers, which is enough to perform two multiplies in parallel and thus hide some of the latency. Given that multiplications are efficient, we now propose a new permutation step for strong hashing.

4.1 Zipper Merge

Intuitively, it is clear that the highest and lowest bits of a multiplication result are more predictable. This is the basis of the (problematic) “middle square” random generator [11], which only retains the middle digits of a multiplication result. We introduce a simple but seemingly novel approach: mixing multiplication results with byte-level permute instructions.

Let us derive a suitable permutation. Recall that inputs are 64-bit multiplication results that will become 32-bit multiplicands in the next update round. It therefore makes sense to concentrate the poorly-distributed top and bottom bytes in the upper 32 bits to ensure the multiplier bytes are uniformly good (requirement 1). To increase mixing, we also wish to interleave bytes from the neighboring SIMD lane, ideally with no more than two adjacent bytes from the same lane (requirement 2). Mostly importantly, we strive to equalize the ‘quality’ of each byte within a 64-bit lane (requirement 3). We approximate this by counting the minimum distance of each bit’s position from the ends of its lane, computing the sum of these distances for the bits in a byte, and sorting these sums in decreasing order. By this measure, bytes 3 and 4 are best, 2, 5, 1, 6 are adequate, and 0 and 7 are worst. Permutation results R_i for a 16 byte lane pair S_i are expressed as hexadecimal offsets P_i, such that $R_i = S_{P_i}$. In other words, the i-th offset indicates which source byte to copy into the i-th result byte. Let $i = 15$ be listed first and $i = 0$ last. A partial permutation satisfying requirements 1 and 2 takes the form `7 8 6 9 ? ? ? ? 0 F 1 E ? ? ? ?`. To see this, note that the source lanes (offset divided by 8) alternate, and that the byte offsets (modulo 8) are `7 0 6 1`, which are the worst as mentioned. It remains to distribute bytes 2-5 and A-D between both lower 32-bit lane halves. `5 2 C 3` and `D A 4 B` is a feasible solution, with no more than two adjacent bytes and exactly equal quality in both lanes. The final permutation is `7 8 6 9 D A 4 B 0 F 1 E 5 2 C 3`. If we partition the inputs into two 64-bit parts and view these as ‘highway lanes’, the bytes (nearly) alternate between these two lanes. We therefore borrow the term ‘zipper merge’ from the road context. Note that this particular permutation is not necessarily optimal – it would be interesting, but computationally expensive, to optimize it based on the resulting hash bit bias (discussed in Section 6).

4.2 Update

The proposed `Update` folds a 32 byte vector of inputs (`packet`) into internal state variables `v*` and `mul*` by multiplying and permuting:

```
v1 += packet;
v1 += mul0;
mul0 ^= V4x64U(_mm256_mul_epu32(v0, v1 >> 32));
v0 += mul1;
mul1 ^= V4x64U(_mm256_mul_epu32(v1, v0 >> 32));
v0 += ZipperMerge(v1);
v1 += ZipperMerge(v0);
```

`V4x64U` is a vector class that provides member functions and overloaded operators, which are more convenient than using SIMD intrinsics (such as `_mm256_mul_epu32`) directly. Note the symmetrical structures, which resemble the butterfly portion of Fast Fourier Transforms (FT). This is reasonable – recursive discrete FT also need to combine or mix their inputs in a similar way. The key operations here are permutations of the input and state, which helps ensure no information about the key leaks (see the analysis in Section 6). Note that the multiplication latency is partially hidden behind other instructions because `mul` are only needed in the next `Update`. There is no need to similarly delay `ZipperMerge` because its latency is just one cycle [10].

4.3 Finalization

After consuming all input data, the internal state must be further mixed to reduce the risk of key leakage. It is convenient to use the same `Update` function. How many rounds are necessary? We hashed all 2^{24} combinations of three input bytes and computed the probability of an output bit toggling in response to an input bit changing. Strong biases remain after two rounds, but decrease by a factor of 300 after three rounds to 0.03%. We added a fourth round for safety, which did not have a measurable impact on the average bias.

To ensure the upper vector lanes are mixed into the final result, we must also permute them between each round. Note that `ZipperMerge` only combines adjacent lanes. The AVX2 instruction set generally does not allow interaction between the 128-bit halves of a vector, presumably to allow reuse of logic from prior 128-bit SSE2 hardware. However, 64-bit lanes can be shuffled at modest latency cost via `_mm256_permutevar8x32_epi32`. We swap the 128-bit vector halves and also 32-bit lane halves of `v0` using the following indices of 32-bit parts: `2 3 0 1 6 7 4 5`. The result is passed to `Update` and this process repeated another three times. Note that the this `Permute` operation will also be used during initialization.

We then add together the four state vectors `v*` and `mul*`, reducing the 1024-bit state into four 64-bit lanes. Mixing via addition is slightly more thorough than XOR because of carry ripples. A 64-bit hash suffices for many applications, in which case we only retain the lower lane, which is slightly easier to extract into general-purpose registers. Note that a quantum algorithm can find collisions in $O(\sqrt[3]{N})$ time [12]. If a hash is to resist such attacks, it should consist of at least 192 bits ($N = 2^{192}$). HighwayHash can produce up to 256-bit hashes at no additional cost; they also pass our test suite. However, our analysis of differential attacks in Section 6.2 only applies to the 64-bit case. HighwayHash needs further strengthening and analysis before it can serve as a cryptographically secure message digest.

4.4 Initialization

We assume 256 key bits are reasonable and sufficient. Attackers have an astronomically low, 1 in 2^{256-s} chance of guessing the key after evaluating 2^s inputs. The key must be expanded fourfold to populate the 1024 bits of internal state. A two-fold expansion is achieved by initializing `v0` with the key XOR a constant, and `v1` with the `Permute`-d key (see Section 4.3) XOR a second constant. To make clear that there is no malicious intent behind the choice of constants, we use "nothing up my sleeve" numbers and document the process. We begin with a hexadecimal representation of the initial digits of π [13]. To ensure each bit is set in at least one of the four lanes, we modify the fourth number by setting each bit if zero or one other lanes have that bit set. Finally, we initialize `mul0, mul1` to the first and second constant, respectively.

4.5 Padding

`Update` operates on entire 32-byte vectors, so inputs must be padded to multiples of that size. We use the same scheme as SipHash [1], described below for completeness. It is important that zero-valued buffers of various sizes have different hashes. The only difference is their length, so we need to include that in the input data. To reduce the likelihood of processing an additional vector, we wish to minimize the added size. We therefore insert the size modulo 256, encoded in a single byte. To avoid potential page faults caused by reading past the end of the input buffer, we conceptually copy the remaining 0-31 bytes into a zero-initialized buffer, insert the length into the most significant byte, load it into a vector and then `Update`.

An architectural limitation motivates a slightly different approach. Stores to memory or even cache involve considerable latency. To speed up subsequent loads, CPUs forward data directly from intermediate store buffers. However, smaller unaligned stores (from copying the remainder bytes) cannot be combined to satisfy a larger load of the entire vector [14]. We instead use the new AVX2 instruction `_mm256_maskload_epi32` to load all remaining 32-bit pieces in the input. By masking off the padding, we avoid reading past the end of the input buffer but still load as much of the input as possible in a single operation. The remaining 0-3 bytes are loaded individually, combined with the length byte and inserted into the rest of the vector. This increases hash throughput by about 10 % for 1 KiB inputs.

4.6 Source Code

Our HighwayHash implementation [9] was published as open source software in March 2016 under the Apache 2 license. Its SIMD operations are expressed using `V4x64U`, a custom AVX2 vector class with overloaded operators. For example, `V4x64U a = b + c` is easier to understand than the equivalent SIMD intrinsic `__m256i a = _mm256_add_epi64(b, c)`. To support older CPUs, we also provide a SSE4.1 variant whose throughput is only about 10% lower.

5 Throughput

We took unusual care in measuring the throughput of SipHash, SipTreeHash and HighwayHash. Each is implemented in C++ using AVX2 intrinsics and compiled with GCC 4.8.4 `-std=c++11 -O3 -mavx2`. The benchmark runs on a single core of a desktop Xeon E5-2690 v3 clocked at 2.6 GHz. We record high-resolution timestamps from the invariant TSC and use fences to ensure the measured code is not reordered by the compiler or CPU. To prevent elision of the benchmark computations, we pass the hash result as input constraints to an empty inline assembly block. This forces the compiler to assume that the result is used because the block allegedly modifies memory. We avoid unrealistic branch prediction by randomly interleaving measurements for various sizes, rather than repeatedly hashing the same input size. Note that the benchmark frequently accesses the same data, which ensures it is cache-resident, so the resulting throughput is an upper bound. However, this is common in benchmarks and can be ignored because the single-core hash throughput is lower than the observed memory bandwidth, and far below the peak bandwidth [15]. To eliminate any outliers due to background activity or thermal throttling, we use a robust estimator (the mode). The resulting measurements have a mean absolute deviation of about 0.2 cycles; we retain the median as the final result. This 'nanobenchmark' performance measuring infrastructure is included in the open-source release [9].

Table 1 lists throughputs for several input sizes in the usual unit of CPU cycles per byte. For 1 KB inputs, SipTreeHash and HighwayHash are 2.1 and 3.8 times as fast as SipHash. Reducing SipHash rounds from 2 per update and 4 during finalization to 1 and 3 also increases its throughput by a factor of 1.2 to 2. HighwayHash is 1.1 to 1.3 times as fast as SipTreeHash13.

Table 1: Cycles per byte for various input sizes [bytes].

Algorithm	8	31	32	63	64	1023
SipHash	17.81	5.52	5.72	3.34	3.45	1.47
SipHash13	13.88	4.00	4.25	2.46	2.46	0.74
SipTreeHash	22.98	6.10	5.80	3.02	3.14	0.61
SSE41HighwayHash	17.38	4.58	4.40	2.37	2.28	0.39
SipTreeHash13	19.53	5.21	4.72	2.52	2.47	0.37
HighwayHash	15.42	4.32	3.79	2.04	2.05	0.34

Throughput generally increases for larger inputs (Figure 1) because the finalization cost is amortized over more data. SipTreeHash is slower than SipHash for smaller inputs because it processes entire 32-byte AVX2 vectors, and also hashes the intermediate results. However, AVX2 SIMD instructions process four 64-bit elements at a time, so SipTreeHash eventually outperforms SipHash. It would be faster still if SIMD instruction sets supported bit rotations, which are currently emulated with three instructions. Note the periodic decreases every 32 bytes in both tree hashes. Inputs are padded to entire vectors, so relative throughput decreases as padding increases. The worst case is a multiple of the vector size; adding the extra length byte requires another vector to be processed.

SipHash was designed to efficiently handle small inputs. The tree hashes are much faster for large inputs because they process 32 bytes at a time. Surprisingly, HighwayHash also outperforms SipHash for small inputs due to

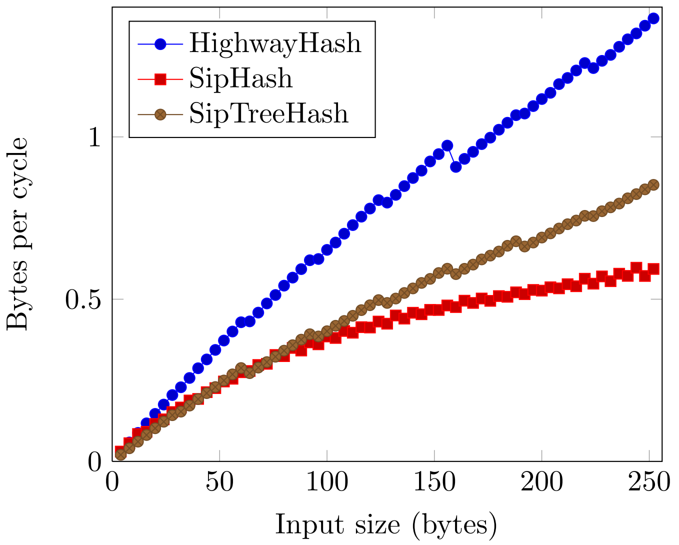

Figure 1: Throughput [bytes per cycle] increases with input size, with dips at multiples of the packet size.

efficient finalization and optimized padding (Section 4.5).

6 Security Analysis

We are not experienced cryptographers, and do not see a way to reduce the HighwayHash algorithm to provably secure constructions. This multiply-permute scheme may require new methods of cryptanalysis. In the absence of a formal proof, we use statistical testing to validate our main claim: HighwayHash is a keyed hash function that is indistinguishable from a uniformly random source, with an avalanche effect comparable to a cryptographic hash.

6.1 smhasher Test Suite

smhasher [16] is a test suite for hash functions that verifies their output distribution and checks for collisions when hashing ‘difficult’ inputs. Recall that CLHash (without additional mixing) fails the avalanche test [6], which requires that half of the output bits change when an input bit is flipped. We further strengthen this test by checking all input sizes between 4 and 32 bytes[1], raising the iteration count from 300K to 900M (at the cost of

[1]Biases computed for 3 bytes are much larger and likely overestimates.

about 5000 CPU-hours for all tests), and using high-quality random numbers. A generator with period p should not be asked to produce more than $\sqrt[3]{p}$ numbers [17]. The input data for all sizes are populated from 78 64-bit random numbers. This implies a minimum period of 2^{109}, which rules out common generators such as XorShift64 and Tausworthe (2^{88}). We instead use the `pcg64_k32` permuted congruential generator [18] with a much larger period of 2^{2176}. The avalanche test is considered successful if each output bit has a bias (deviation from the expected 50% bit flip rate) of less than 1%. FarmHash 1.1's `Fingerprint64`[2] [19], a recent fast non-cryptographic hash, results in 62-77% bias for 25-32 byte inputs. This may enable key recovery or collision attacks. The other hashes have less than 1% bias (Table 2).

Table 2: Avalanche bias: percent difference between ideal and actual bit-flip probability for 4-32 byte keys (lowest value per column <u>underlined</u>).

Algorithm	Lowest	Average	Highest
SipHash	0.0117	<u>0.0131</u>	0.0162
Blake2bp	0.0115	0.0134	0.0156
SipHash13	<u>0.0109</u>	0.0133	0.0153
HighwayHash	0.0117	0.0133	<u>0.0147</u>

Interestingly, the Blake2bp cryptographic hash [20] has slightly higher minimum, average and maximum bias than the faster hashes, and SipHash13 has lower minimum and maximum bias than SipHash. HighwayHash has nearly the same average bias as Blake2bp and SipHash13, but the narrowest range and lowest maximum.

In addition to empirical verification, we discuss several possible attacks and whether they apply to HighwayHash. Our attack model assumes that the secret key is initially unknown.

6.2 Differential Attack

Let us begin by attempting a differential attack. Assume we start in a randomized state, i.e. `v0, v1` are uniformly distributed random variables, which is holds true if the key was random. We will attempt to modify sequential vectors A, B and C such that the state change caused by A is reversed by B. The two rounds of `Update` have the following effect:

[2] `Hash64WithSeed` mixes more thoroughly, but its API contract does not guarantee unchanging results.

```
 1  v1 += A;
 2  v1 += mul0;
 3  mul0 ^= V4x64U(_mm256_mul_epu32(v0, v1 >> 32));
 4  v0 += mul1;
 5  mul1 ^= V4x64U(_mm256_mul_epu32(v1, v0 >> 32));
 6  v0 += ZipperMerge(v1);
 7  v1 += ZipperMerge(v0);
 8  v1 += B;
 9  v1 += mul0;
10  mul0 ^= V4x64U(_mm256_mul_epu32(v0, v1 >> 32));
11  v0 += mul1;
12  mul1 ^= V4x64U(_mm256_mul_epu32(v1, v0 >> 32));
13  v0 += ZipperMerge(v1);
14  v1 += ZipperMerge(v0);
15  v1 += C;
```

The last step is included in case C can help restore the value of `v1`. The effect of A is to either change `mul0` in line 3, or `mul1` in line 5, except for a 1 in 2^{32} chance that the 32 bits from `v0` are 0. In that case, `v1` can probably be corrected in line 8, but we have still changed `v0`, which cannot be corrected before propagating to `mul0` or `mul1` with all but 1 in 2^{32} chance, and the differential attack fails because we are no longer able to influence `mul0` or `mul1`. The combined probability that both changes from A and B do not affect either `mul` value is at most 1 in 2^{64}, so that approach also fails. The other case is where we allow `mul0` or `mul1` to change in line 3 or 5, hoping it will be corrected by the corresponding line 10 or 12. The probability of this is 2^{-32}, with a higher likelihood when we change only 1 bit in A and then B, and lower when we change more bits. In line 3 or 5, there will be at least a 32-bit random change in either `mul0` or `mul1`, which propagates to `v0` or `v1` in line 9 or 11. Correcting `v0` or `v1` is at best another 1 in 2^{32} chance in lines 13-15, and again the attack fails. Note that this analysis holds even in the first vector where the `mul0` and `mul1` values are known, but `v0` and `v1` are random. Longer differentials that include a subsequent vector are too late to help because changing one or more bit in A will avalanche from adding/XORing/multiplying at least 64 bits of random initial state at line 1 from `v0 v1 mul0 mul1` into each other after line 11.

6.3 Length Extension Attacks

Length extension attacks are infeasible because the calls to `PermuteAndUpdate` during finalization differ from previous calls to `Update`. In particular, the permutation involves neighboring lanes, whereas regular `Update` invocations do not. Also, the state is random, depending on the secret key, and permutes in a manner that avalanches to new states on each call to `Update`, regardless of the vector values. In cases where the input length is not a multiple of the vector size (32 bytes), the padding scheme ensures that messages with various numbers of trailing zero bytes result in different hashes.

6.4 Entropy Loss

When all input vectors except one are held constant, the final state will differ for all 2^{256} possible vector values, because `Update` is a permutation. Compression only occurs when injecting input data and discarding all but the 64 output bits during finalization.

6.5 Rotational Attacks

The security of an ARX (Add, Rotate, XOR) scheme S depends on the number of additions q [21]. For n-bit inputs I and a rotation function R we have

$$Z = P(R(x + y, r) = R(x, r) + R(y, r)) = (1 + 2^{r-n} + 2^{-r} + 2^{-n}).$$

$$P[S(R(I, r)) = R(S(I), r)] = Z^q$$

For an ARX scheme with too few additions, we can detect non-randomness in the function by showing that $S(R(I, r)) = R(S(I), r)$ more than a random function, the probability of which is 2^{-n} for an n-bit hash function. For example, SipHash (with 2 update and 4 finalization rounds) involves 13 additions. The resulting state values, when XORed together, will be the same whether we rotate the inputs by 32 or just the output by 32 with probability $4^{-13} = 2^{-26}$. If keys are reused and attackers choose the input, a significant bias should be detectable. However, it is not clear how this can be used to either create collisions or reveal the key. Direct rotational attacks to not seem to apply to the multiplier, since the output is 64-bits whereas each operand is 32 bits. However, each multiplier contributes on the average 16 32-bit additions, and the four finalization rounds include 8 multiplications for a total average of 128 32-bit additions. For the results to be equal after rotating both inputs by 16, we need the upper 16 bits of one operand to be 0, and the lower 16 bits of the other to be 0, with probability 2^{-32}. The chance of this happening throughout all finalization rounds is negligible. With a total of 32 multiplications (4 lanes times 8), the resulting 1 in 4^{1024} bias is far too small to detect.

6.6 SAT Solvers

Boolean satisfiability solvers have been applied to the problem of deriving unknown key bits when the attacker knows most bits of the key in SipHash. Modified versions that use fewer finalization rounds were found to be vulnerable to modern SAT solvers. However, formal verification of CPU multiplier circuits remains a challenge. Polynomial time solutions for verification have been discovered, but they rely on algebraic properties of multiplication and cannot be used on larger circuits also including XOR gates. We therefore expect HighwayHash to be far more resistant to SAT-solver based key extraction than traditional ARX hash functions such as SipHash.

7 Conclusion

Faster hashing could save enormous amounts of CPU time in data centers. However, algorithms entirely focused on speed, including Murmur3 and CityHash, are vulnerable to attacks [3]. For example, adversaries can skew the distribution of 'random' decisions. We describe a strengthened version of the smhasher test suite [16] that reveals a previously unknown weakness in `Fingerprint64` from FarmHash 1.1 [19] (but not `Hash64WithSeed`, which is fast and well-distributed). Applications should not use such unsafe hash functions unless they trust their input data. SipHash is a widely used pseudo-random function for which no relevant attacks are known. However, it is relatively slow (1.3 cycles per byte for 1 KB inputs). We propose HighwayHash, a novel keyed hash that is almost 4 times as fast thanks to SIMD multiply and permute instructions. The source code is available under the permissive Apache 2 license at `https://github.com/google/highwayhash`. To the best of our current (non-expert) ability, we have analyzed the algorithm and found no weaknesses. Statistical tests indicate HighwayHash is less predictable than SipHash. We also considered powerful attacks, including differential and rotational, and are confident the algorithm will withstand these specific attacks. We welcome further cryptanalysis, especially because hash functions built using multiplication and permutation have not yet been studied. Assuming this construction remains unbroken, it can be recommended for pseudo-random number generators and fast message authentication codes. Strengthened variants may also substantially accelerate stream ciphers and file hashing. Such sensitive applications should incorporate provably secure primitives into the finalization, while still benefiting from the fast HighwayHash `Update` function.

References

[1] J. Aumasson and D. Bernstein. Siphash: a fast short-input PRF. *IACR*, 2012: 351, 2012. URL http://eprint.iacr.org/2012/351.

[2] ECRYPT. Measurements of hash functions, indexed by machine, September 2015. URL https://bench.cr.yp.to/results-hash.html.

[3] M. Bolet J. Aumasson, D. Bernstein. Hash-flooding DoS reloaded: attacks and defenses, December 2012. URL https://131002.net/siphash/siphashdos_29c3_slides.pdf.

[4] S. Gueron and Y. Lindell. GCM-SIV: Full nonce misuse-resistant authenticated encryption at under one cycle per byte. *IACR*, 2015:102, 2015. URL http://eprint.iacr.org/2015/102.

[5] V. Hoang, T. Krovetz, and P. Rogaway. AEZ v4.1: Authenticated encryption by enciphering, October 2015. URL http://web.cs.ucdavis.edu/~rogaway/aez/aez.pdf.

[6] D. Lemire and O. Kaser. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering*, 6(3):171–185, 2016. URL http://dx.doi.org/10.1007/s13389-015-0110-5.

[7] D. Lemire. Private communication, December 2016.

[8] Shay Gueron. Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256. *IACR*, 2014:170, 2014. URL http://eprint.iacr.org/2014/170.

[9] J. Wassenberg. Fast strong hash functions: Siphash/Highwayhash, May 2016. URL https://github.com/google/highwayhash.

[10] A. Fog. Instruction tables, January 2016. URL http://agner.org/optimize/instruction_tables.pdf.

[11] J. von Neumann. Various techniques for use in connection with random digits. In *von Neumann's Collected Works*, volume 5, pages 768–770. Pergamon, 1963.

[12] G. Brassard, P. Hoyer, and A. Tapp. Quantum algorithm for the collision problem, May 1997. URL http://arxiv.org/abs/quant-ph/9705002.

[13] A. Yee. Pi, September 2015. URL http://www.numberworld.org/digits/Pi/.

[14] Intel. Intel 64 and IA-32 architectures optimization reference manual, January 2016. URL http://goo.gl/9IkxGj.

[15] Intel. Intel Xeon processor E5-1650 v3. URL http://ark.intel.com/products/82765/Intel-Xeon-Processor-E5-1650-v3-15M-Cache-3_50-GHz.

[16] A. Appleby. SMHasher is a test suite, January 2016. URL https://github.com/aappleby/smhasher.

[17] P. L'Ecuyer and R. Simard. On the performance of birthday spacings tests for certain families of random number generators. *Mathematics and Computers in Simulation*, February 2001. URL https://goo.gl/BVfPFu.

[18] M. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation, December 2014. URL http://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf.

[19] G. Pike. Farmhash, a family of hash functions, March 2015. URL https://github.com/google/farmhash.

[20] J. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. Blake2 official implementations, 2016. URL https://github.com/BLAKE2/BLAKE2.

[21] D. Khovratovich, I. Nikolic, J. Pieprzyk, P. Sokolowski, and R. Steinfeld. Rotational cryptanalysis of ARX revisited. *IACR*, 2015:95, 2015. URL http://eprint.iacr.org/2015/095.