Modern Hash Function and Pseudorandom Number Generator

Yi Wang[1,2], Diego Barrios Romero[3], Daniel Lemire[4], Li Jin[1,2*]

1 Ministry of Education Key Laboratory of Contemporary Anthropology, Collaborative Innovation Center for Genetics and Development, School of Life Sciences, Shanghai, China.

2 Human Phenome Institute, Fudan University, Shanghai, China.

3 Fraunhofer Institute for Digital Medicine MEVIS, Bremen, Germany.

4 Université du Québec (TÉLUQ) Montreal, Canada

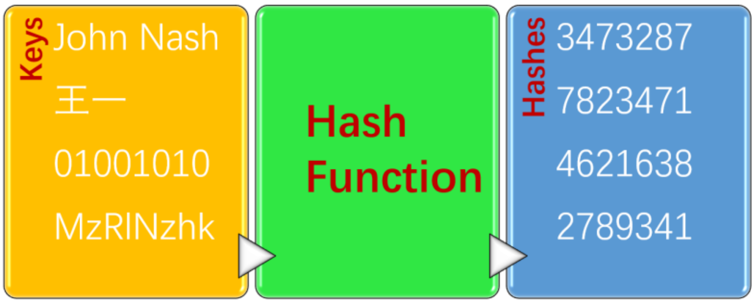
ABSTRACT

Hash function and pseudorandom number generator (PRNG) are two fundamental functions in computer science with numerous applications. Due to their importance, hundreds of hash functions and PRNGs have been proposed in last decades. However, there is still no consensus non-cryptographic hash function and PRNG that possess both quality, speed, simplicity and portability. We propose wyhash and wyrand as modern hash function and PRNG respectively in hope to update corresponding standard library functions. They are of high quality and portable across 32bit/64bit, little/big endian and aligned/unaligned architectures. Benchmark and user feedback suggest a significant speedup by simply replacing existing hash functions and PRNGs with them. Now they have been packed into Debian software source and become the default of the V and Zig language. wyhash and wyrand are completely free under The Unlicense at https://github.com/wangyi-fudan/wyhash.

INTRODUCTION

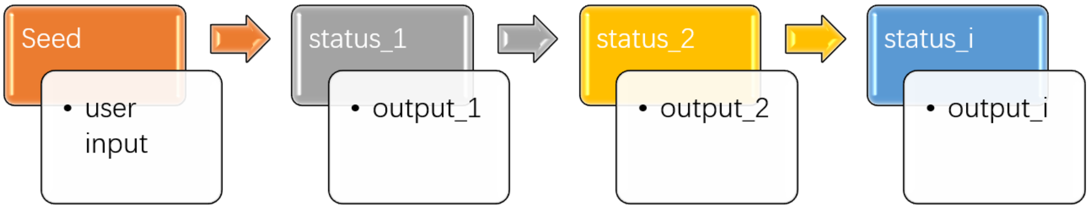
A hash function is a function that convert arbitrary data to fixed-size hash values (usually integers) [1] (Figure1). The input data was called the "keys" and the output was called the "hashes". Hash function is a cornerstone of computer science and has numerous applications: hash table, bloom filters, authentication code [1], file checksum, duplication/collision detection [2], proof-of-work [3], etc. [4].

Figure 1: Illustration of hash function



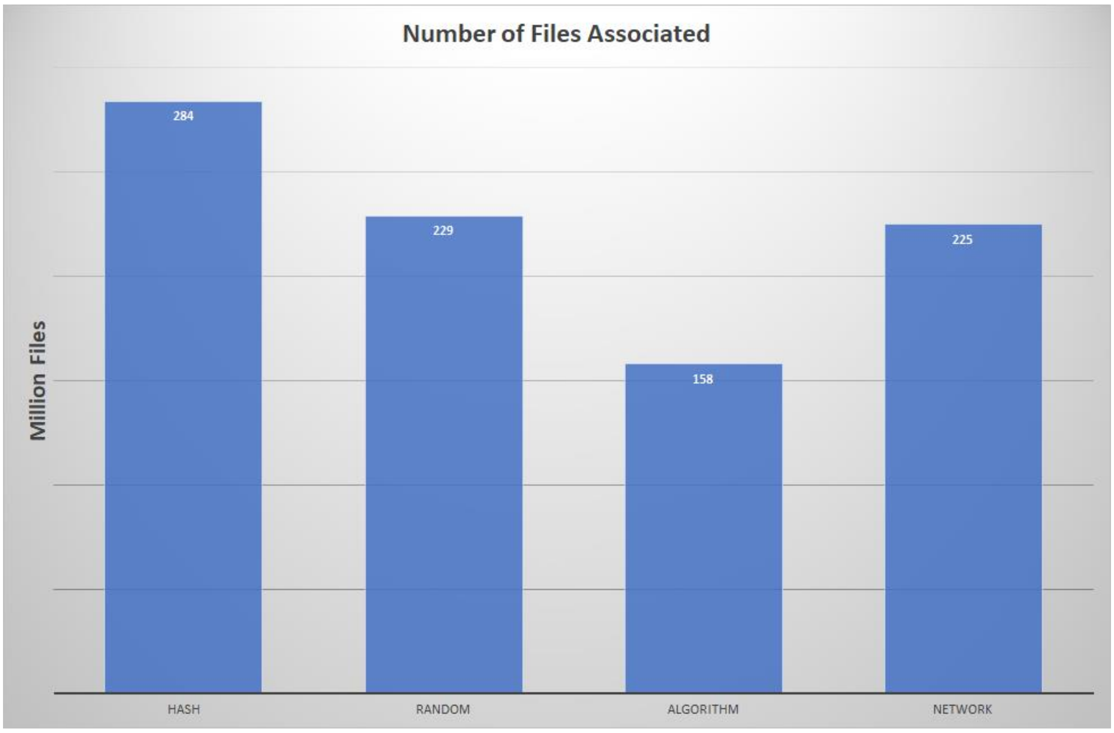
A pseudo-random number generator (PRNG) is an algorithm that can generate a stream of numbers which appears random (Figure 2). The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the "seed". [5] PRNG brings "randomness" to a deterministic computer, thus has wide applications: randomized algorithm [6], statistical sampling [7], simulation [8], gaming etc. [4].

Figure 2: Illustration of pseudo-random number generator



To roughly illustrate the popularity of hash function and PRNG in computer science, we searched GitHub. Figure 3 shows the number of GitHub files that associated with the several keywords respectively. Surprisingly, "hash" and "random" is as popular as "algorithm" and "network", where the later two are well known to be key importance in the computer world. Due to their popularity and hence importance, numerous hash functions [9] and PRNGs [10][11][12] have been designed in last decades.

Figure 3: Number of GitHub files that contain keywords



Despite the richness of hash functions and PRNGs, there is still no consensus non-cryptographic hash function and PRNG that possess both quality, speed, simplicity and portability [9-12]. The quality of hash function and PRNG is characterized by their uniformity and independence of output distribution [9-12]. It is the premise of hash function [27] and PRNG and can be evaluated by SMHasher [9], PractRand [11] and BigCrush [12]. The speed is characterized by the number of function calls per second or Gigabytes processing per seconds. In practice short key hashing speed attracts more attention as real key length distribution is biased to short ones [13]. Simplicity is measured by number of instructions after compilation [9]. Simple hash function and PRNG are not only reduce cache footprint but also aesthetically amusing. In practice we also require portability which means the hash function and PRNG should support different machine architectures such as 32-bit/64-bit, little/big endian, aligned/unaligned memory etc.

To approach such ideal hash function and PRNG, we propose wyhash and wyrand [14] with one year of continuous development. They are of high quality that pass SMHasher, PractRand and BigCrush. They are the fastest conventional hash function and PRNG at the premise of high quality. Their code sizes are small. They are portable to both 32-bit/64-bit, little/big endian, aligned/unaligned machine architectures. Considering the possession of these advantages, we bravely name wyhash and wyrand the "modern" non-cryptographic hash function and PRNG respectively in the hope to update corresponding standard library ones. wyhash and wyrand are open sourced and were distributed under The Unlicense [15] which means completely free.

RESULT

Quality Validation

We perform statistical quality test on wyhash by SMHasher [9]. wyhash passed all quality tests. (SI: SMHasher.wyhash.txt). We performed statistical quality test of wyrand by PractRand [11] and BigCrush [12] via testingRNG suite [10]. wyrand passed all tests (SI: PractRand.wyrand.log, testwyrand-b.log, testwyrand-r-b.log, testwyrand-z-b.log).

Hashing Speed Benchmark

According to SMHasher, the following 16 out of 174 hash functions are 64-bit quality and portable hash functions: *poly_2_mersenne, poly_3_mersenne, poly_4_mersenne, tabulation, floppsyhash, SipHash, GoodOAAT, prvhash42_64, HighwayHash64, mirhashstrict, pengyhash, FarmHash64, farmhash64_c, t2ha_atonce, xxHash64, wyhash.*

We benchmarked all these functions plus the std::hash with SMHasher which contains the bulk speed test, short key speed test and hash map speed test. Figure 4 shows the bulk hash speed of hash functions. Wyhash is the fastest one which is as 3.2X fast as std::hash. Figure 5 shows the small key hash cycles. Wyhash has the lowest cycles per hash which is as 2.3X fast as std::hash. Figure 6 shows the hash map cycles. Wyhash is the fastest one which is as 1.6X fast as std::hash.

Figure 4: Bulk Hashing Speed Benchmark

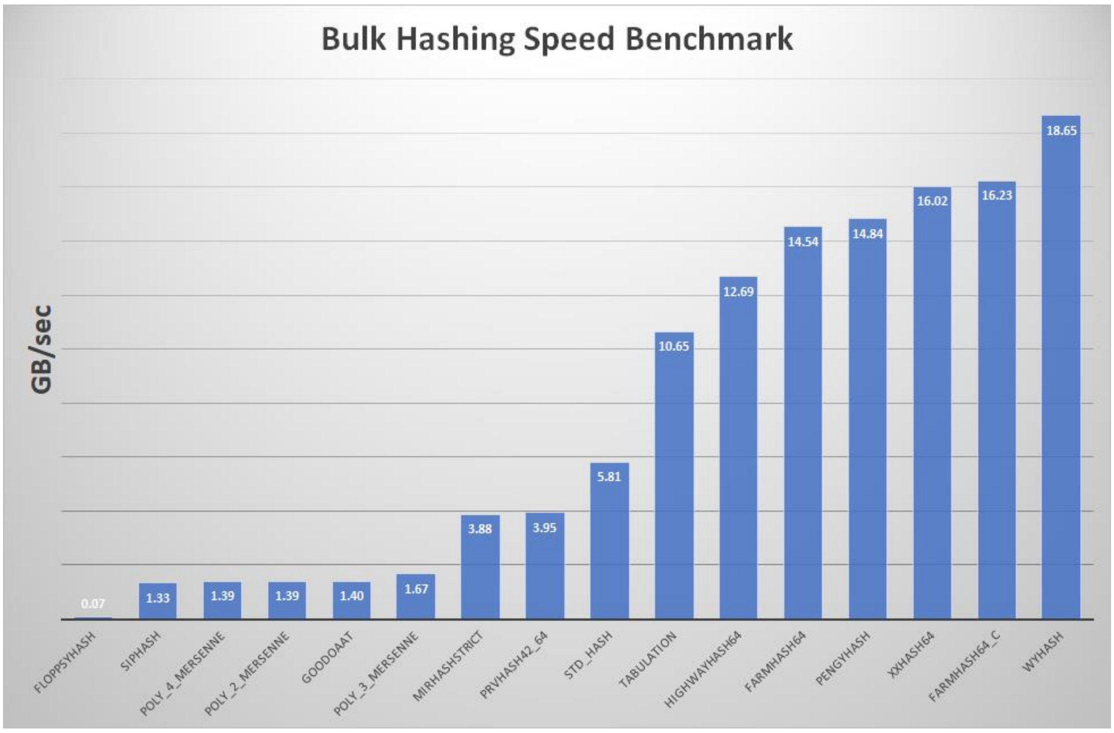


Figure 5 : Short Key Hash Cycles

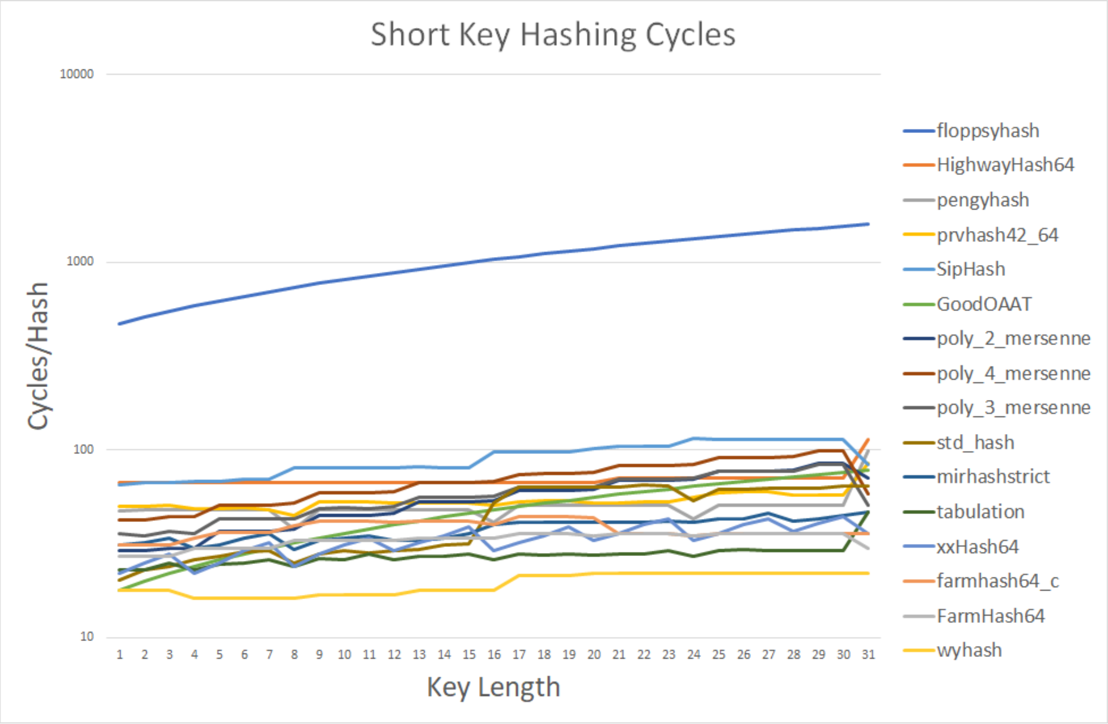
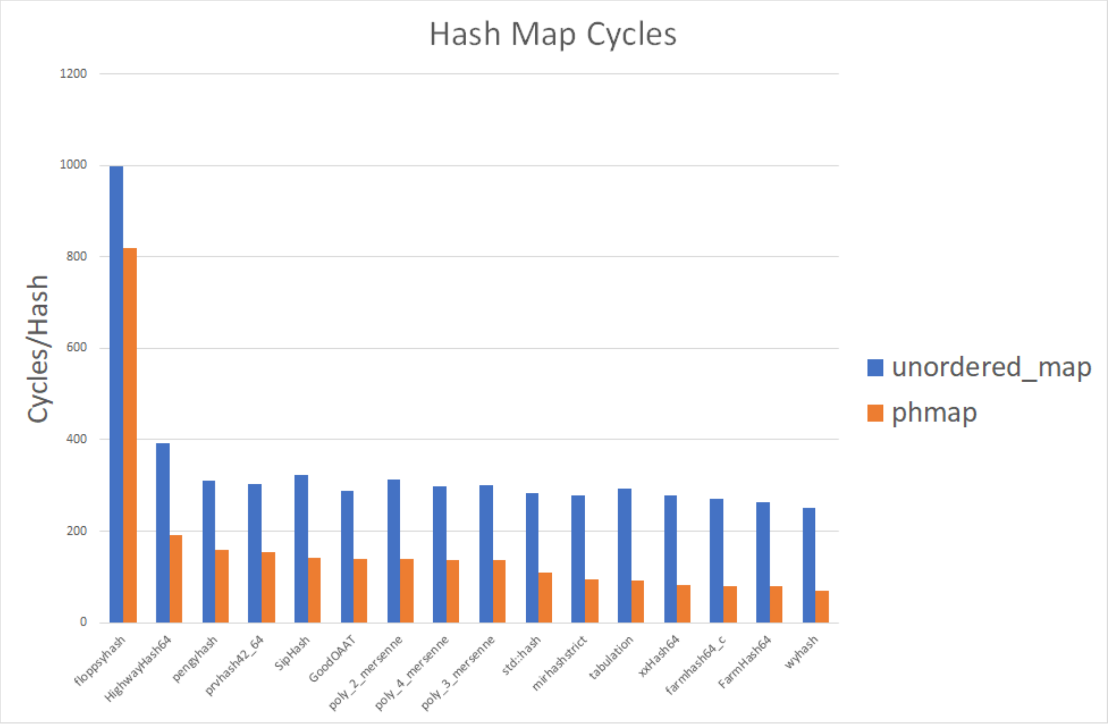


Figure 6: Hash Map Cycles

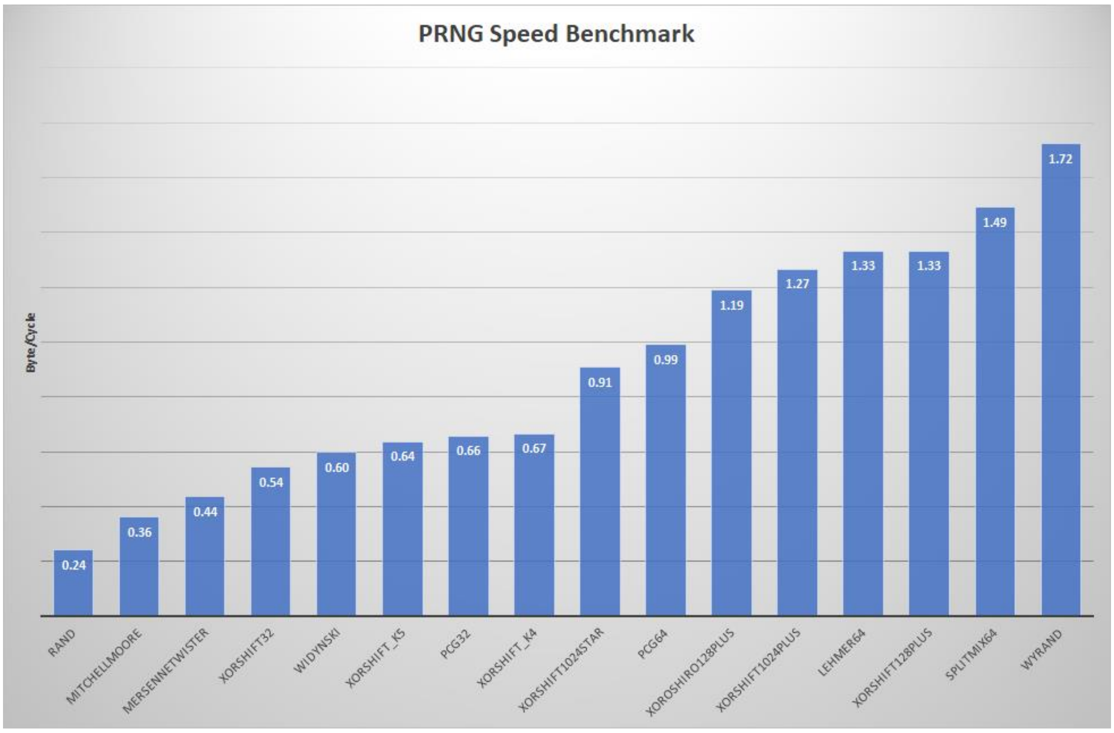


PRNG Speed Benchmark

We benchmarked all portable PRNG in testingRNG suite: xorshift_k4, xorshift_k5, mersennetwister, mitchellmoore, widynski, xorshift32, pcg32, rand, lehmer64, xorshift128plus, xoroshiro128plus, splitmix64, pcg64, xorshift1024star, xorshift1024plus, wyrand.

Figure 7 shows the PRNG speed benchmark result. We observe that wyrand is the fastest one which is as 7.2X fast as the C library function rand, and as 3.9X fast as the famous Mersenne Twister [24].

Figure 7: PRNG Speed Benchmark



Code Size Comparison

We obtain compiled code size of 64-bit quality and portable hashes from SMHasher home page [9]. FigureS1 shows the comparison of code size. Wyhash is at median of code size distribution which is reasonably small. Wyrand code size is also minimal which is documented in SI.

User Feedback

After 18 months of exposure to public, wyhash and wyrand have already gained 271 stars and certain impacts on downstream applications. They have become the default for the V [16] and Zig language [17]. For the V language wyhash become a game changer which make its hash map faster than B-tree implementation [18]. Remote desktop software xorgxrdp got 3X speedup on 4K screen latency by simply replacing CRC hash function with wyhash [19]. Microsoft HoloLens project becomes “much faster” on X86 CPU by switching to wyhash [20]. Mergerfs avoids crashing on some architectures by replacing fasthash64 with wyhash [21].

Conclusion

Based on these results, we conclude that wyhash and wyrand are high quality, fast, simple and portable modern hash function and PRNG respectively. We expect significant speedup by simply replacing library hash function and PRNG with the modern wyhash and wyrand.

DISCUSSION

The core function underlying wyhash and wyrand is the MUM function: MUM (A, B) -> C, where A, B, C are 64-bit unsigned integers [online Method]. As @leo-yuriev pointed out [25], MUM function without xoring mask is vulnerable, as MUM (0, X) =0 for any X which losses entropy. As a solution to this problem, we evolved to the masked-MUM=MUM (A^secret, B^seed). By keeping the mask as secrets or randomized value, masked-MUM cannot be cracked trivially in non-cryptographic applications. However, in rare case (2^{-64}), A^secret=0 or B^seed=0 is still possible. Further protection against such cases is also available at some cost of speed by defining a higher security level and invoke the secure-MUM (A, B) =MUM (A, B) ^A^B. It is obvious that for A=0, secure-MUM (A, B) =B will not loss entropy.

Wyrand uses 64-bit internal status and produce 64-bit output. This function is not bijective [26]. However, it is not necessary to worry about its quality because (1) it has passed strict statistical test and (2) bijective is even not a good property for a PRNG. Image we have a smaller PRNG which has 8-bit internal status and a bijective 8-bit output. When we draw an output, we will be sure that this number will never come again within next 255 draws due to the bijective constrain. Thus, bijective PRNG violates the randomness expectation and is not a good property for a PRNG.

Wyhash use memcpy to access memory safely. It does not do unaligned memory access which is unsafe on some machines. Despite the nominal overhead of memcpy calls, it is actually as fast as direct memory read thanks to the complier optimization. By default, wyhash does not depend on the “read through” method that read across memory bound. However, in particular cases where the short key hashing speed is of critical importance, wyhash can use such method and doubling short key hashing speed by defining a lower security level.

ACKNOWLEDGEMENTS

We sincerely thank the following names due to their contributions to wyhash development: Reini Urban, Dietrich Epp, Joshua Haberman, Tommy Ettinger, Otmar Ertl, cocowalla, leo-yuriev, Diego Barrios Romero, paulie-g, dumblob, Yann Collet, ivte-ms, hyb, James Z.M. Gao, Devin.

REFERENCES

1 Daniel Lemire, Owen Kaser: Faster 64-bit universal hashing using carry-less multiplications. Journal of Cryptographic Engineering Volume: 6, Issue: 3, pp 171-185 (2016) DOI: 10.1007/S13389-015-0110-5

2 R. Rivest: The MD5 Message-Digest Algorithm. The MD5 Message-Digest Algorithm Volume: 1321, pp 1-21 (1992)

3 Melanie Swan: Blockchain: Blueprint for a New Economy (2015)

4 https://github.com/

5 Andrew Rukhin ,Juan Soto ,James Nechvatal ,Miles Smid ,Elaine Barker: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Special Publication (NIST SP) - 800-22 Rev 1a (2000) DOI: 10.6028/NIST.SP.800-22R1A

6 Rajeev Motwani,Prabhakar Raghavan: Randomized Algorithms.(1994)

7 Joseph Felsenstein: CONFIDENCE LIMITS ON PHYLOGENIES: AN APPROACH USING THE BOOTSTRAP. Evolution Volume: 39, Issue: 4, pp 783-791 (1985) DOI: 10.1111/J.1558-5646.1985.TB00420.X

8 M. P. Allen 1,D. J. Tildesley: Computer Simulation of Liquids (1988)

9 https://github.com/rurban/smhasher

10 https://github.com/lemire/testingRNG

11 Doty-Humphrey C (2010) Practically random: C++ library of statistical tests for rngs. https:// sourceforge.net/projects/pracrand

12 L'Ecuyer P, Simard R (2007) Testu01: Ac library for empirical testing of random number generators. ACM Trans Math Soft (TOMS) 33(4):22

13 https://github.com/rurban/perl-hash-stats

14 https://github.com/wangyi-fudan/wyhash

15 https://unlicense.org/

16 https://github.com/vlang/v

17 https://github.com/ziglang/zig

18 https://github.com/vlang/v/pull/3591

19 https://github.com/neutrinolabs/xorgxrdp/pull/167

20 https://github.com/microsoft/MixedReality-Sharing/issues/115

21 https://github.com/trapexit/mergerfs/pull/805

22 https://github.com/vnmakarov/mum-hash

23 https://github.com/Cyan4973/xxHash

24 M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998) DOI:10.1145/272991.272995

25 https://github.com/wangyi-fudan/wyhash/issues/49

26 https://github.com/wangyi-fudan/wyhash/issues/16

27 Martin Dietzfelbinger: On Randomness in Hash Functions. Symposium on Theoretical Aspects of Computer Science Volume: 14, pp 25-28 (2012)