Modern Non-Cryptographic Hash Function and Pseudorandom Number Generator

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

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

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

3 Université du Québec (TÉLUQ) Montreal, Canada

*Corresponding Author:
Li Jin: lijin@fudan.edu.cn

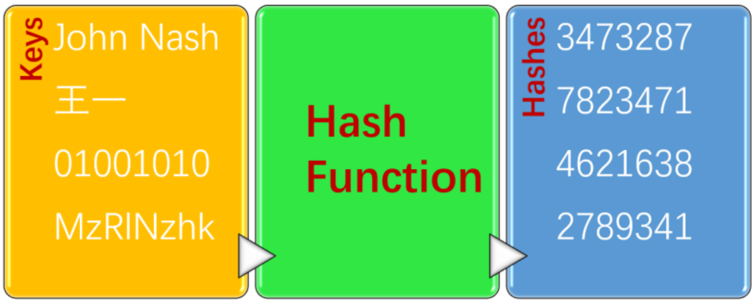
ABSTRACT

Hash function and pseudorandom number generator (PRNG) are two fundamental functions in computer science with numerous applications. Due to their popularity and importance, hundreds of hash functions and PRNGs have been proposed in last decades. However, few of non-cryptographic hash functions and PRNGs achieve both quality, speed, portability and simplicity to reach a new consensus beyond the standard library functions. Here, we propose wyhash hash function and wyrand PRNG as modern alternatives to the decades-old standard library functions. They are of high quality and portable across 32bit/64bit, little/big endian and aligned/unaligned architectures as well as VisualC++/gcc/clang compilers. Benchmark and user feedback suggest a significant speedup by simply replacing existing hash functions or PRNGs with them. Now they have been packed into Debian software source and become the default of the Nim, V and Zig language. wyhash and wyrand are completely free under The Unlicense at https://github.com/wangyi-fudan/wyhash.

INTRODUCTION

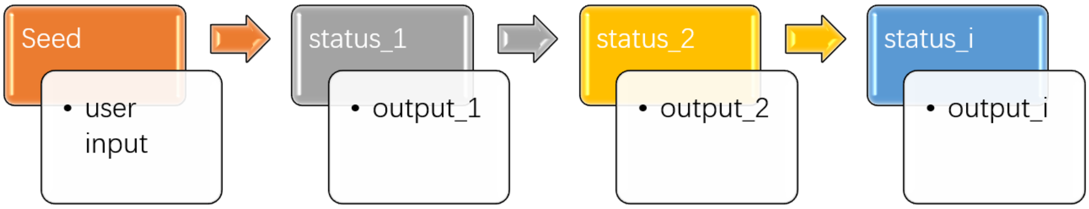
A hash function is a function that convert arbitrary data to fixed-size hash values which are usually integers [1] (Figure1). The input data was called the "keys" and the output was called the "hashes". Hash function is a cornerstone of computer science and has numerous applications: hash table, bloom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

Figure 1: Illustration of hash function



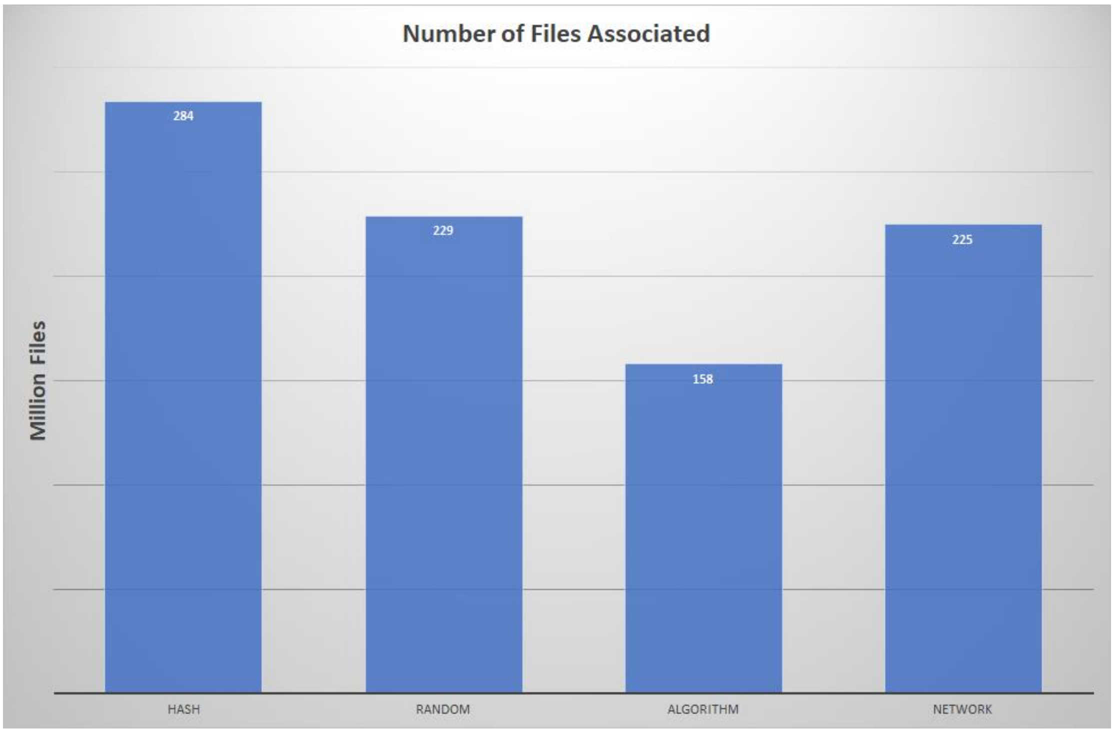
A pseudo-random number generator (PRNG) is an algorithm that can generate a stream of numbers which appears random (Figure 2). The PRNG-generated sequence is not truly random, because it is completely determined by an initial value provided by the user, called the "seed". [5] PRNG enbles a deterministic computer with "randomness" 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 to broader audience, we searched GitHub [4]. 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 [10-12] have been designed in last decades as alternatives to the standard library functions.

Figure 3: Number of GitHub files that contain keywords



Despite the richness of hash functions and PRNGs, few of non-cryptographic hash functions and PRNGs achieve both quality, speed, portability and simplicity to reach a new consensus beyond the standard library functions. [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], BigCrush [12] and dieharder[29]. The speed is the main goal at the promise of quality. In practice short key hashing speed attracts more attention as real key length distribution is biased to short ones [13]. We also emphasize portability which means the hash function and PRNG should support as many machine architectures and compilers as possible. Simplicity is measured by number of instructions of the function after compilation [9]. Simple hash function and PRNG are not only cache efficient but also aesthetically amusing.

To approach a new consensus on non-cryptographic hash function and PRNG, we introduce wyhash hash function and wyrand PRNG [14]. They are of high quality that pass SMHasher, PractRand, BigCrush and dieharder. They are the fastest conventional hash function and PRNG at the premise of high quality. They are portable to both 32-bit/64-bit, little/big endian, aligned/unaligned machine architectures as well as VisualC++/gcc/clang compilers. Their code sizes are small and were distributed under The Unlicense [15] which means completely free. Considering these advantages, we bravely propose them as modern alternatives to the decades-old low-quality standard library functions [9-12].

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] as well as dieharder [29]. wyrand passed all tests (SI: PractRand.wyrand.log, testwyrand*.log wyrand*.txt).

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, the short key speed test and the Hashmap speed test. Figure 4 shows the bulk hash speed of hash functions. Wyhash is the fastest one which is as 3.9X 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

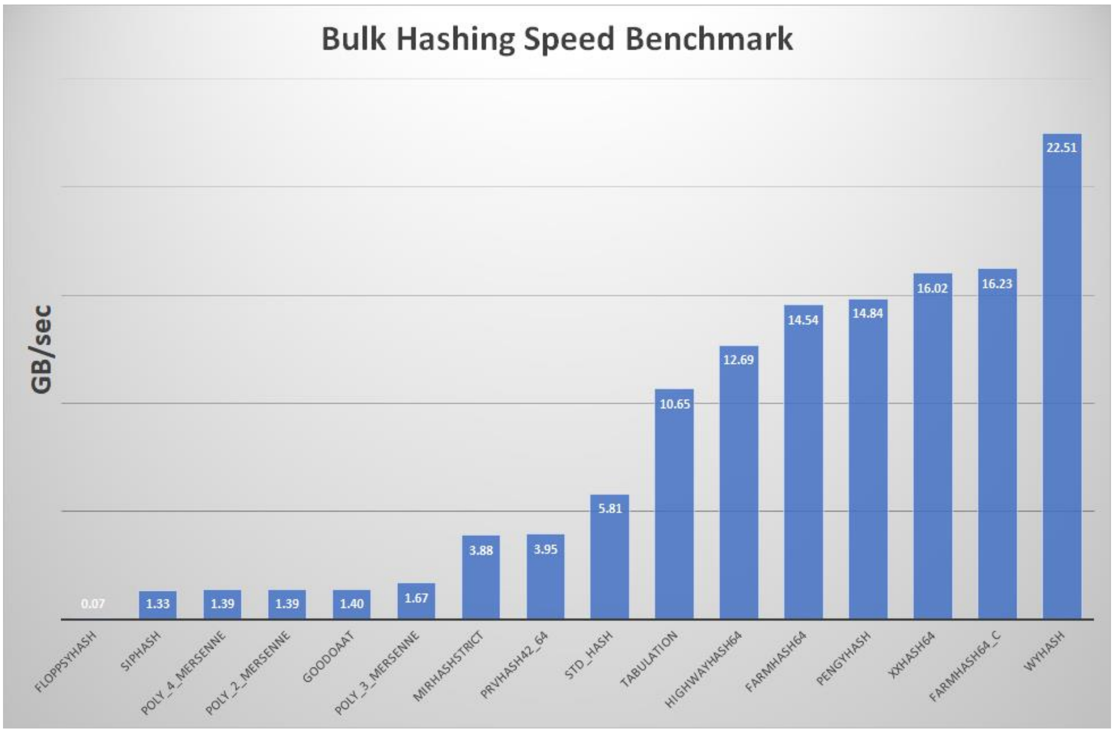


Figure 5 :   Short Key Hash Cycles

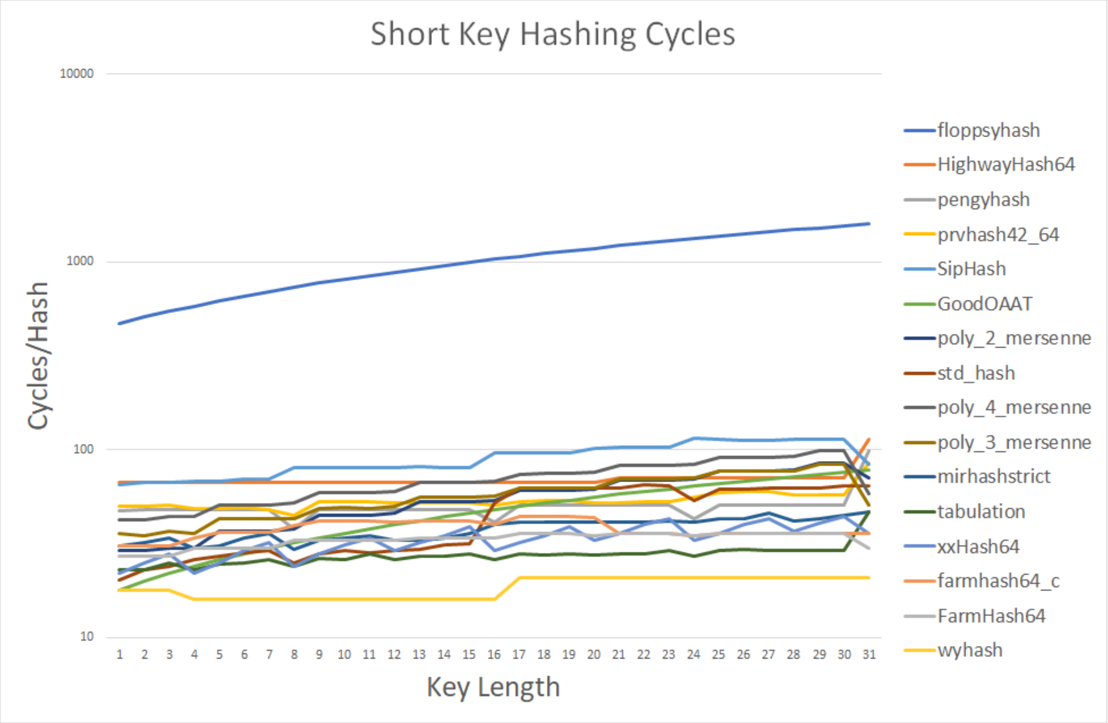
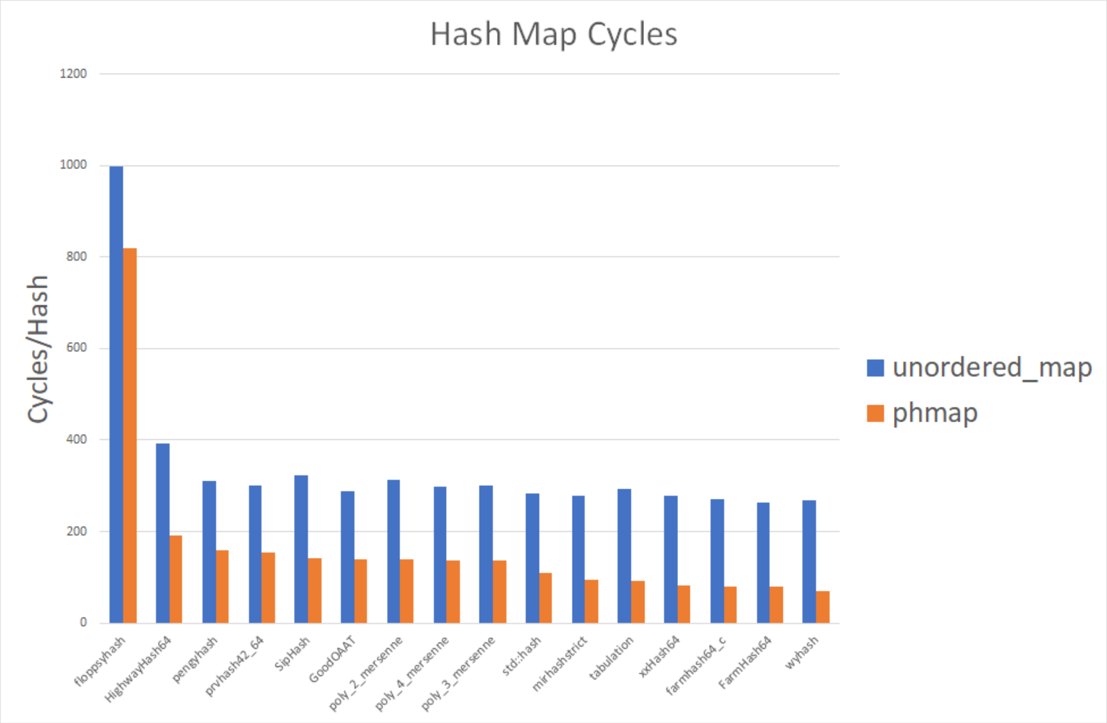


Figure 6: Hash Map Cycles

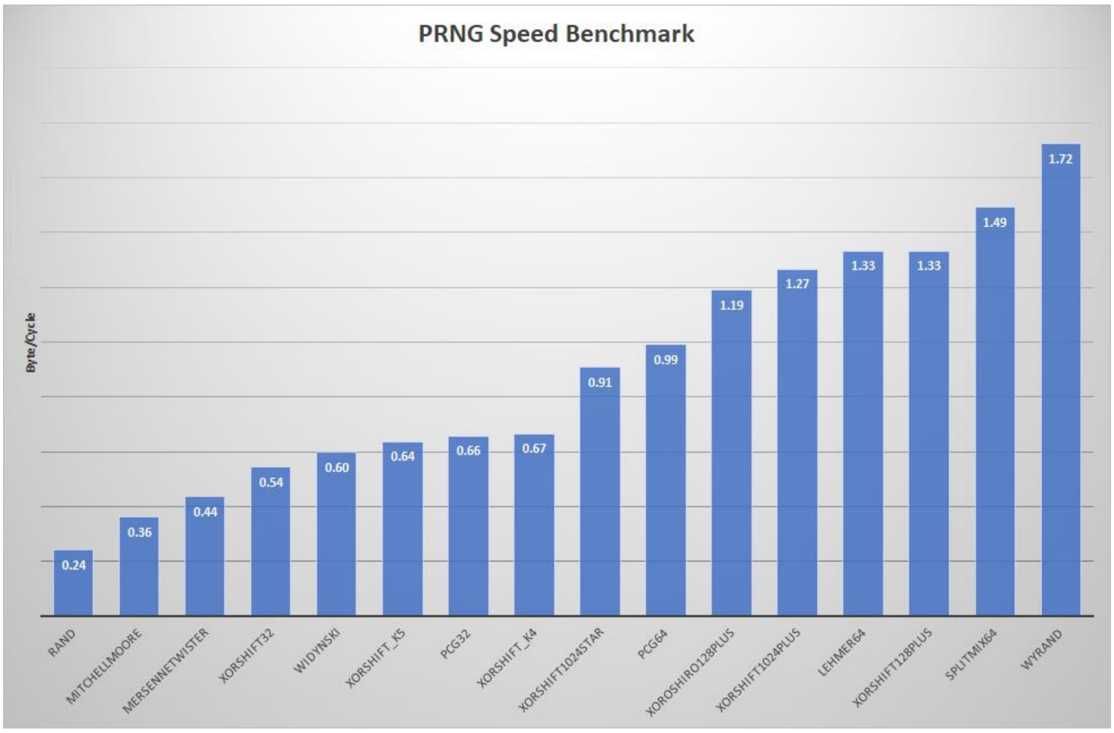


PRNG Speed Benchmark

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



Portability

Wyhash and wyrand are portable to 32-bit/64-bit, little/big endian, aligned/unaligned memory architectures as well as VisualC++/gcc/clang compilers due to defines contributed by many user feedbacks.

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 small end of code size distribution. Wyrand code size is also minimal which is documented in SI.

User Feedback

After two year of exposure to public, wyhash and wyrand have already gained 420 stars and rich impacts on downstream applications. They have become the default for the Nim [28], 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, fastest, portable and simple hash function and PRNG respectively. User can expect a significant speedup in hash/PRNG heavy tasks by simply replacing existing functions with them. Considering these advantages, we call for broader application of them and suggest standardizing them to be modern alternatives to standard library functions.

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 loses 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 lose 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. Imagine 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.

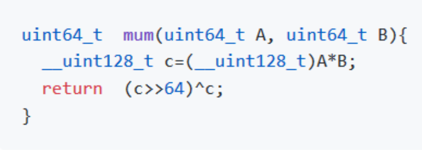
Wyhash use memcpy to access memory safely. It does not do unaligned memory access which is unsafe on some machines. Despite the nominal overhead of memcpy calls, it is actually as fast as direct memory read thanks to the compiler optimization.

Supplement Information

METHOD

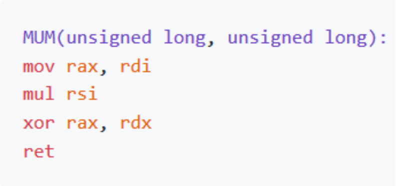
Mix Function:

Wyhash and wyrand is based on a mix function call MUM that mix two 64-bit integer A and B to produce a 64-bit integer C: MUM (A, B) => C. @vnmakarov released the original version of MUM on Mother's Day [22].



```
uint64_t  mum(uint64_t A, uint64_t B){
  __uint128_t c=(__uint128_t)A*B;
  return  (c>>64)^c;
}
```

Despite the nominal 128-bit multiplication, the actual instructions on 64-bit machines are as simple as follow:

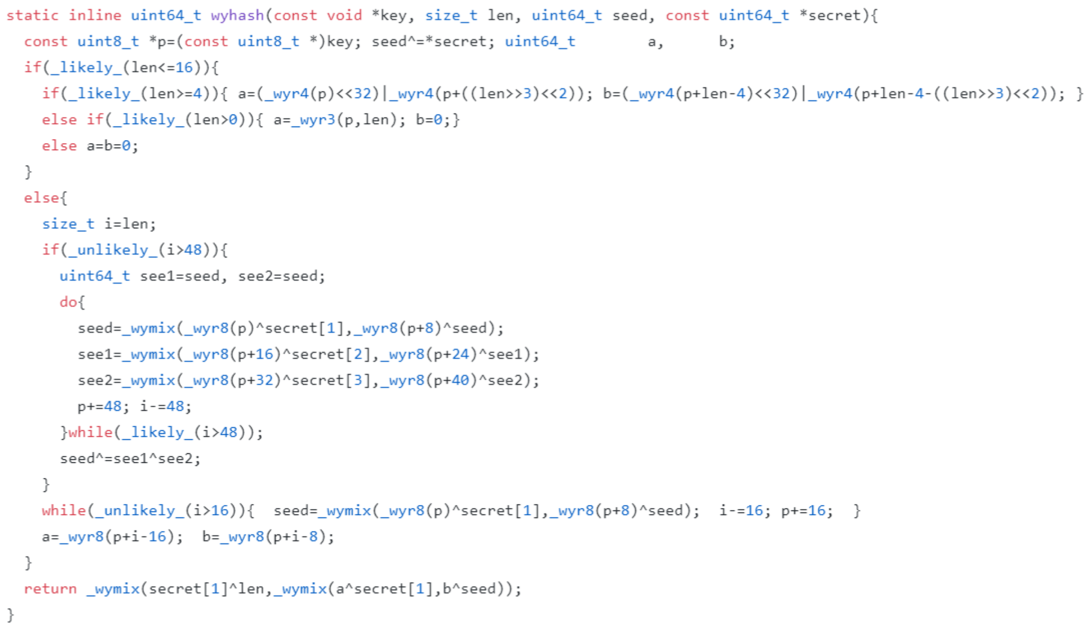


```
MUM(unsigned long, unsigned long):
mov rax, rdi
mul rsi
xor rax, rdx
ret
```

Our further improvements on MUM is the masked-MUM: MUM (A^secret, B^seed), where secret is a predefined 64-bit integer with 32 bits set and seed is status with a uniform distributed number of 1bits. The masked-MUM can protect the MUM from being zero (Discussion), randomize the distribution of real data and produce an avalanche effect. We observed experimentally that just two rounds of masked-MUM suffice to pass all statistical tests.

wyhash Hash Function

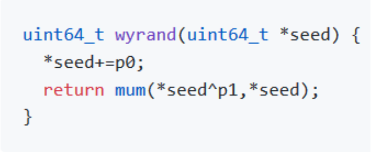
wyhash hash function is based on masked-MUM and contains three parts: the 16 bytes part, the batch part and the finalization part. If the data is less then 16 bytes, it will be processed by the 16 bytes part and the finalization part. Otherwise it will be processed by the batch part and the finalization part. The key iteration is seed=MUM (8byte-data1^secret, 8byte-data2^seed). The code is shown below where the _wyr# functions reads # bytes from the key using memcpy and _wymix is the alias of MUM function.



```c
static inline uint64_t wyhash(const void *key, size_t len, uint64_t seed, const uint64_t *secret){
  const uint8_t *p=(const uint8_t *)key; seed^=*secret; uint64_t        a,      b;
  if(_likely_(len<=16)){
    if(_likely_(len>=4)){ a=(_wyr4(p)<<32)|_wyr4(p+((len>>3)<<2)); b=(_wyr4(p+len-4)<<32)|_wyr4(p+len-4-((len>>3)<<2)); }
    else if(_likely_(len>0)){ a=_wyr3(p,len); b=0;}
    else a=b=0;
  }
  else{
    size_t i=len;
    if(_unlikely_(i>48)){
      uint64_t see1=seed, see2=seed;
      do{
        seed=_wymix(_wyr8(p)^secret[1],_wyr8(p+8)^seed);
        see1=_wymix(_wyr8(p+16)^secret[2],_wyr8(p+24)^see1);
        see2=_wymix(_wyr8(p+32)^secret[3],_wyr8(p+40)^see2);
        p+=48; i-=48;
      }while(_likely_(i>48));
      seed^=see1^see2;
    }
    while(_unlikely_(i>16)){  seed=_wymix(_wyr8(p)^secret[1],_wyr8(p+8)^seed);  i-=16; p+=16;  }
    a=_wyr8(p+i-16);  b=_wyr8(p+i-8);
  }
  return _wymix(secret[1]^len,_wymix(a^secret[1],b^seed));
}
```

wyrand PRNG

Our PRNG named wyrand is even simpler. It keeps a 64-bit internal status and updates it by adding a 64-bit prime. The internal status is mixed with masked itself by MUM function to produce a pseudorandom number. It is obvious that its cycle length is 2^{64} as p0 is a large prime.

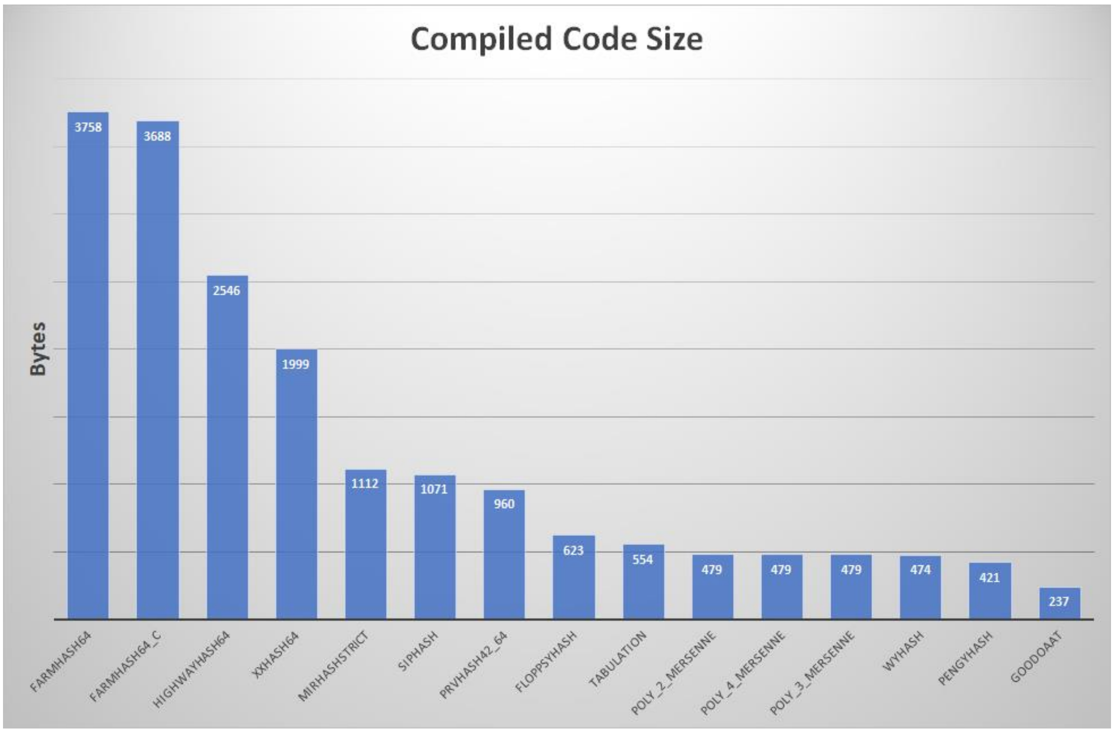


```c
uint64_t wyrand(uint64_t *seed) {
  *seed+=p0;
  return mum(*seed^p1,*seed);
}
```

Benchmark

We validate and benchmark wyhash and wyrand on a server with 2X Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz, 64GB memory and 2*2TB SSD hard driver. SMHasher [9] is used to validate and benchmark hash functions. PractRand [11] and BigCrush [12] in testingRNG [10] test suite is used to validate wyrand. testingRNG is used for benchmark PRNGs.

FigureS1: Compiled Code Size Hash Functions



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)

28 https://github.com/nim-lang/Nim

[29] https://github.com/rurban/dieharder