| | |
|---|---|
| Paper Number: | P2075R1 |
| Title: | Philox as an extension of the C++ RNG engines |

Authors: Pavel Dyakov <pavel.dyakov@intel.com>
Ilya Burylov <ilya.burylov@intel.com>
Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>
Andrey Nikolaev <Andrey.Nikolaev@intel.com>
John Salmon <John.Salmon@DEShawResearch.com>

| | |
|---|---|
| Audience: | SG6 (Numerics), LEWG |
| Date: | 2020-07-15 |

## I. Introduction

C++11 introduced a comprehensive mechanism to manage the generation of random numbers in the <random> header file (including distributions, pseudo random and non-deterministic engines).

We proposed a set of engine candidates for the C++ standard extension in the P1932R0 paper [1]. This paper is focused on the family of the counter-based Philox engines.

We propose 2 possible API approaches and seek feedback from the committee on which path is preferable.

## II. Revision History

Key changes compared with R0 (reviewed in Prague in SG6):

- Aligned wording for philox_engine with the C++ standard.
- Added an alternative API in Section VIa with a std::array template parameter. Removed alternative APIs with calculated constant values.
- Added an alternative approach in Section VII with a generic counter_based_engine and a specific philox_prf pseudo-random function.

## III. Motivation

See P1932R0 [1] for motivation.

## IV. General Description

Philox is one of the counter-based engines introduced in 2011 in [2]. All counter-based engines have a small state (e.g., Philox4x32 has 10 x 32-bit elements in its state) and a long period (e.g., the period of Philox4x32 is $2^{130}$). Counter-based engines effectively support parallel simulations via both block-splitting and independent-stream techniques and many of them (including Philox) are well-suited to a wide variety of hardware including CPU/GPU/FPGA/etc.

Philox is proposed as the first new standardized engine since C++11. It satisfies the following criteria, as discussed in P1932R0 [1]):

- **Statistical properties.** The original paper asserted that the Philox family passes rigorous statistical tests including hundreds of different invocations of TestU01's BigCrush [2]. This statement has been independently confirmed: the TestU01 batteries for Philox4x32-10 and Philox4x32-7 were tested in [4] and DieHard testing results for Philox4x32-10 were published in the Intel® Math Kernel Library (Intel® MKL) documentation [5].
- **Wide usage.** Philox is broadly used in Monte-Carlo simulations which require massively parallel random number generation, e.g., financial simulations [6], simulation of non-deterministic finite automata [7], etc.

- **HW friendliness.** Philox's distinguishing features are its small state and reliance on simple primitive operations. It is, therefore, easy to vectorize and parallelize. On a CPU, for example, Intel® MKL provides a highly vectorized version of Philox4x32-10. Philox is also proven to work on GPUs – it's implemented in the GPU-optimized Nvidia and AMD libraries: cuRAND and rocRAND.

## V. Proposed API and Wording

Two approaches to an API definition are investigated:

1) A philox-focused API defines a self-contained engine class template analogous to the other random number engines in the standard. No new concepts are introduced, only one new engine class template and a small number of pre-defined philox_NxW template aliases. (This is an evolution of the R0 version of this paper).
2) A counter-based-engine API which is more generic and permits/encourages the creation of engines based on other pseudo-random functions as well. The pre-defined philox_NxW template aliases are instantiations of counter_based_engine in much the same way that knuth_b is a pre-defined instantiation of the shuffle_order_engine adaptor.

## VI. Philox-Focused API and Wording

This API specifies a single, new philox_engine class template. Pre-defined aliases are provided for instantiations with constants and parameters that are known to produce high-quality random numbers.

The philoxNxW aliases have a pre-defined round-count, r=10, that is somewhat larger than the minimum required to pass known statistical tests. In other words, they provide a statistical safety margin at a modest performance cost. The philoxNxW_r<r> permit the program to trade speed for safety by specifying a different number of rounds of mixing. Philox generators with r=7 have no known statistical flaws [2].

The philox_engine is described in terms of the Philox function which acts as a keyed bijection on a domain of size $2^{W*N}$. Consequently, the philoxNxW engines have a period of $N*2^{W*N}$.

The changes affect only section "26.6 Random number generation".

- ***Changes in section 26.6 Random number generation***

…

(5.3) – the operator mullo denotes the low half of the modular multiplication of a and b: $(a * b) mod\ 2^{\boldsymbol{w}}$

(5.4) – the operator mulhi denotes the high half of the multiplication of a and b: ( $\lfloor (a * b)/2^{\boldsymbol{w}} \rfloor$ )

- ***Changes in sub-section 26.6.1 Header <random> synopsis***

…

// 26.6.3.4 *class template* philox_engine

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType
         ...consts>
    class philox_engine;
```

…

// 26.6.5 engines and engine adaptors with predefined parameters.

…

```
template<int r>
using philox4x32_r<r> = see below;

template<int r>
using philox4x64_r<r> = see below;

using philox4x32 = philox4x64_r<10>;

using philox4x64 = philox4x64_r<10>;
```

…

- ***New sub-section "26.6.3.4 Class template philox_engine"***

## 26.6.3.4 Class template philox_engine

1 A `philox_engine` random number engine produces unsigned integer random numbers in the closed interval $[0, 2^w - 1]$, where the template parameter *w* defines the range of the produced numbers. The state $x_i$ of a `philox_engine` object is of size *(5n/2+1)* and consists of a sequence *X* of *n* result_types, a sequence *K* of *n/2* result_types, a sequence *Y* of n result_types, and a scalar, *I*, the index of the next value to be returned by the GA from *Y*.

2 The generation algorithm GA($x_i$) returns $Y_I$, the value stored in the $I^{th}$ element of *Y*, in state $x_{i+1}$, i.e., **after** applying the transition algorithm: $x_{i+1}$ = TA($x_i$).

3 The state transition algorithm, TA, is performed as follows:

```
I=I+1
if(I == n){
    Y = Philox(K, X)    // see below
    X = (X+1)  //as if X is an n*w-bit integer
    I = 0
}
```

4 The Philox function maps the n/2-length sequence K and the n-length sequence X into an n-length output sequence. Philox applies an *R*-round substitution-permutation network to the values in X. Each round consists of a P-box that permutes all n values, and *n/2* S-boxes that operate on two values at a time [2]. A single round of the generation algorithm performs the following steps:

(4.1) – The output sequence X' of the previous round (*X* in case of the first round) is passed to the P-box. For *n>2*, the P-box performs the following permutation to obtain the intermediate state V:

$$V_j = X'_{f(j)}$$

where $j = 0, \dots, n-1$ and $f(j)$ is defined in Table 1 below, as in [2, 9]:

*Table 1. Values for the word permutation f(j)*

| | | j= | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| n = | 4 | 0 | 3 | 2 | 1 | | | | | | | | | | | | |
| | 8 | 2 | 1 | 4 | 7 | 6 | 5 | 0 | 3 | | | | | | | | |
| | 16 | 0 | 9 | 2 | 13 | 6 | 11 | 4 | 15 | 10 | 7 | 12 | 3 | 14 | 5 | 8 | 1 |

[*Note*: for *n=2* the P-box performs no permutation]

(4.2) – Consecutive elements of the V sequence, $V_{2*k}$ and $V_{2k+1}$, are passed to the k-th S-box, $k = 0, \dots, n/2 - 1$. Each S-box performs the computation:

$$X'_{2*k} = mullo(V_{2*k+1}, M_k)$$
$$X'_{2*k+1} = mulhi(V_{2*k+1}, M_k)\ xor\ key_k^r\ xor\ V_{2*k}$$

where: q is the index of the round: q = 0 … $r-1$, k is the index of the S-box: k = 0 … n/2-1, $key_k^q$ is the kth round key for round q, $key_k^q = (K_k + q * C_k)\ mod\ 2^w$, and $M_k$ and $C_k$ are constants (template parameters).

5 After r applications of the single-round function, Philox returns the value of X'.

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType
        ...consts>
class philox_engine {
    // Exposition only
    static constexpr std::size_t array_size = n / 2;

public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr std::size_t word_size   = w;
    static constexpr std::size_t word_count  = n;
    static constexpr std::size_t round_count = r;
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }
    static constexpr result_type default_seed = 20111115u;

    // constructors and seeding functions
    philox_engine() : philox_engine(default_seed) {}
    explicit philox_engine(result_type value);
    template<class Sseq> explicit philox_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

6 The template parameter `…consts` represents the $M_k$ and $C_k$ constants which are grouped per S-box (M_k, C_k) where k is index of S-box, as follows: $[M_0, C_0, M_1, C_1, M_2, C_2\ \dots\ M_{N/2-1}, C_{N/2-1}]$

7 The following relations shall hold: $(n\ ==\ 2)\ ||\ (n\ ==\ 4)\ ||\ (n\ ==\ 8)\ ||\ (n\ ==\ 16), 0\ <\ r$, $w <= numeric\_limits < UIntType >:: digits, n\ ==\ sizeof \dots (consts)$.

8 The textual representation of $x_i$ consists of the values of $K_0, \dots, K_{n/2-1}, X_0, \dots, X_{n-1}$ and *I*, in that order. Note that the stream extraction operator can reconstruct *Y* from *K* and *X*, as needed.

```
explicit philox_engine(result_type value);
```

9 *Effects:* Sets the $K_0$ element of sequence $K$ to value. All elements of sequences $X$ and $K$ (except $K_0$) are set as 0. The value of *I* is set to n-1.

```
template<class Sseq> explicit philox_engine(Sseq& q);
```

10 *Effects*: With $W = \lceil w/32 \rceil$ and *a* an array (or equivalent) of length $(n/2) * W$, invokes q.generate(a+0, a+n/2*W) and then iteratively for i=0, …, $n/2-1$, sets $K_i$ to $\left(\sum_{j=0}^{W-1} a[i * W + j] * 2^{32*j}\right) mod 2^w$. All elements of sequence $X$ are set to 0. The value of *I* is set to n-1.

- ***Changes in sub-section 26.6.5 Engines and engine adaptors with predefined parameters***

...

```
template<size_t r>
using philox4x32_r = philox_engine<uint_fast32_t, 32, 4, r, 0xD2511F53, 0x9E3779B9,
0xCD9E8D57, 0xBB67AE85>;
```

1 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox4x32_r<10>` produces the value XXXXXXXXX

```
template<size_t r>
using philox4x64_r = philox_engine<uint_fast64_t, 64, 4, r, 0xD2E7470EE14C6C93,
0x9E3779B97F4A7C15, 0xCA5A826395121157, 0xBB67AE8584CAA73B>;
```

2 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox4x64_r<10>` produces the value XXXXXXXXX

```
using philox4x32 = philox4x32_r<10>;

using philox4x64 = philox4x64_r<10>;
```

*[Note*: `philox4x32` and `philox4x64` define the most broadly used Philox parameter sets (supported in Intel® MKL, rocRAND, cuRAND, MATLAB, etc.)*]*

*Other possible pre-defined aliases*:

```
template<size_t r>
using philox2x32_r = philox_engine<uint_fast32_t, 32, 2, r, 0xD2511F53, 0x9E3779B9>;
```

1 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox2x32_r<10>` produces the value XXXXXXXXX

```
template<size_t r>
using philox2x64_r = philox_engine<uint_fast64_t, 64, 2, r, 0xD2B74407B1CE6E93,
0x9E3779B97F4A7C15>;
```

2 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox2x64_r<10>` produces the value XXXXXXXXX

```
using philox2x32 = philox2x32_r<10>;

using philox2x64 = philox2x64_r<10>;
```

[*Note*: `philox2x32` and `philox2x64` do not appear to be broadly used but still show good statistical properties and performance [8].]

### a. Possible Alternative Philox-focused APIs

The template parameter `consts` from the API described in Section V can also be represented as a std::array.

```
// ***************************************************************************
// Alternative API: consts template parameter represented as std::array
// ***************************************************************************

template<typename UIntType, std::size_t w, std::size_t n, std::size_t r,
std::array<UIntType, n> consts>
class philox_engine {
    static constexpr std::size_t array_size = n / 2; // Exposition only

public:
…
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;
…
```

Other alternative APIs were considered but due to received feedback they were eliminated.

## VII. Generic counter_based_engine API

An alternative specification divides the philox engine into 2 entities:

- A pseudo-random function, philox_prf, defined as a class template, which encapsulates the logic contained in the Philox function (but not the TA or GA) in section VI.
- A `counter_based_engine` class template, which encapsulates the TA and GA described in section VI, but depends on a generic pseudo-random function template parameter to generate randomness. Instantiations of counter_based<philox_prf> result in engines with exactly the same properties as the philox_engines described in section VI.

This approach requires slightly more standardized machinery, e.g., a pseudo_random_function concept to constrain the permissible values of the counter_based_engine's template parameter, but it paves the way for a variety of engines with desirable properties. For example, the Threefry engine mentioned in P1932R0 as a candidate for standardization and engines based on widely deployed pseudo-random functions such as SipHash [10] and Chacha [11] are easily accommodated. Furthermore, programmers can, relatively easily, implement new pseudo-random functions with desirable properties for specific purposes (perhaps trading quality or bit-width for speed or size), instantiate a counter_based_engine and immediately gain access to the full power of <random>.

### a. Class template philox_prf

A pseudo-random function (PRF) is a stateless function-like class that returns an array of unsigned integer values when invoked with an array of unsigned integer values. The Philox function specified in the description of the TA in section VI above is just such a function. For the counter-based API, it is hoisted out of the philox_engine and given an independent existence as a class template.

The philox_prf class template may be declared as follows:

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType
          ...consts>
class philox_prf {
    // Exposition only
    static constexpr std::size_t s_box_count = n / 2;
public:
    // generic PRF characteristics: types, data and function members
    using result_type = UIntType;
    static constexpr std::size_t word_size    = w;
    static constexpr std::size_t input_count  = 3 * s_box_count;
    static constexpr std::size_t output_count = n;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }

    // Philox specific characteristics
    static constexpr std::size_t round_count  = r;
    static constexpr std::array<UIntType, s_box_count> multipliers;
    static constexpr std::array<UIntType, s_box_count> round_consts;

    // generic signature of generating function
    template<typename InputIterator1, typename OutputIterator2>
    void operator()(InputIterator1 input, OutputIterator2 output);
};
```

The philox_prf's member `operator()(InputIterator1 input, OutputIterator2 output)` method acts as follows:

1. Copy exactly n/2 values from `input` into sequence K, as if by doing $K_i$ =*(input++); for i in 0,…,n/2-1, in order.

2. Copy exactly n values from `input` into sequence X, as if by doing $X_i$ = *input++; for i in 0,…,n-1, in order.
3. Perform the steps described above in Section VI for the Philox(K, X) function.
4. Copy exactly n values of the Philox function's final value of X' to `output`, as if by doing *(output++) = $X'_i$; for i in 0,…,n, in order

The `philox_prf` has predefined instantiations analogous to those of the philox engine, above:

```
// PRF for R-round Philox with output consisting of 4 32-bit words
template<int R>
using philox4x32_r_prf = philox_prf<uint_fast32_t, 32, 4, R, 0xD2511F53, 0x9E3779B9,
0xCD9E8D57, 0xBB67AE85>;

// PRF for R-round Philox with output consisting of 4 64-bit words
template<int R>
using philox4x64_r_prf = philox_prf<uint_fast64_t, 64, 4, R, 0xD2E7470EE14C6C93,
0x9E3779B97F4A7C15, 0xCA5A826395121157, 0xBB67AE8584CAA73B>;

// PRF for 10-round Philox with output consisting of 4 32-bit words
using philox4x32_prf = philox4x32_r_prf<10>;

// PRF for 10-round Philox with output consisting of 4 64-bit words
using philox4x64_prf = philox4x64_r_prf<10>;
```

Pseudo-random functions are stateless, pure functions. So it makes no sense to state the value of the 10000th invocation. Instead, the standard will state the values returned by a specific invocation, e.g.,

With Z={0x243f6a8885a308d3, 0x13198a2e03707344, 0xa4093822299f31d0, 0x082efa98ec4e6c89, 0x452821e638d01377, 0xbe5466cf34e90c6c} , philox4x64_prf(Z) shall return an array containing:

{0xa528f45403e61d95, 0x38c72dbd566e9788, 0xa5a1610e72fd18b5, 0x57bd43b5e52b7fe6}

With Z={0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344, 0xa4093822, 0x299f31d0}, philox4x32_prf(Z) shall return an array containing:

{0xd16cfe09, 0x94fdcceb, 0x5001e420, 0x24126ea1}

N.B. these values are from the known-answer-test "kat_vectors" in the reference implementation of philox [8].

### b. The pseudo_random_function concept

The `philox_prf` class template has a small number of public constexpr values (input_count, output_count, word_size), dependent class types (result_type) and static public member functions (min(), max()). These members are required of any class that is intended for use as a pseudo-random function by counter_based_engine and will be formalized as a `pseudo_random_function` concept.

### c. Class template counter_based_engine

Instantiations of the class template `counter_based_engine` satisfy the requirements of a random number engine. The result_type, the word_size, and min() and max() functions are obtained from the first template parameter, `prf`, which is constrained to satisfy the requirements of a `pseudo_random_function`. The second template parameter, `size_t c`, specifies how many of the prf's input values are used for the engine's sequence counter. The period of the resulting engine is thus prf::output_count * $2^{c*\text{prf::word\_size}}$.

The specifications here are very similar to those in the "philox-focused" API above, with only minor differences arising because various sequence lengths and constants are obtained from the prf template parameter.

1 A `counter_based_engine<prf, c>` is a random number engine producing unsigned integer values of type result_type = prf::result_type in the closed interval $[0, 2^{prf::word\_size} - 1]$. The state $x_i$ of a `counter_based_engine` object is of size *(*prf::input_count + prf::output_count*+1)* and consists of a sequence *Z* of prf::input_count result_types a sequence *Y* of prf::output_count result_types and an index, *I*, of the next value to be returned by the GA from *Y*. For exposition purposes, the sequence *Z* is treated as the concatenation of a sequence, *K*, of $N_K$=(prf::input_count-c) result_types, and a sequence, *X*, of c result_types. I.e.,

$Z = [K_0\ K_1 \ldots K_{Nk-1}\ X_0\ X_1 \ldots X_{c-1}]$.

In the descriptions that follow, assignments to elements of *X* and *K* are understood as assignments to the corresponding elements of *Z*.

2 The generation algorithm GA($x_i$) returns $Y_I$, the value stored in the $I^{th}$ element of *Y*, in state $x_{i+1}$, i.e., **after** applying the transition algorithm: $x_{i+1}$ = TA($x_i$).

3 The TA is:

```
I=I+1
If(I == prf::output_count){
    prf{}(begin(Z), begin(Y))
    X = (X+1) // as if X is a c*prf::word_size-bit integer
    I = 0
}
```

4 The textual representation of $x_i$ consists of the values of $Z_0, \ldots, Z_{prf::input\_count-1}$, and *I*, in that order. Note that the stream extraction operator can reconstruct *Y* from *Z*, as needed.

```
explicit philox_engine(result_type value);
```

5 *Effects:* Sets the $K_0$ element of sequence $K$ to value. All elements of sequences $X$ and $K$ (except $K_0$) are set as 0. The value of *I* is set to (prf::output_count-1).

```
template<class Sseq> explicit philox_engine(Sseq& q);
```

6 *Effects*: With $W = \lceil w/32 \rceil$ and *a* an array (or equivalent) of length $N_K * W$, invokes q.generate(a+0, a+$N_K$*W) and then iteratively for i=0, …, $N_K - 1$, sets $K_i$ to $\left(\sum_{j=0}^{W-1} a[W * i + j] * 2^{32*j}\right) mod 2^w$. All elements of sequence $X$ are set to 0. The value of *I* is set to (prf::output_count-1).

7 The following relations shall hold: c > 0, c < prf::input_count.

```
template<pseudo_random_function prf, size_t c>
class counter_based_engine {
public:
    // types
    using result_type = typename prf::result_type;

    // engine characteristics
    static constexpr std::size_t state_count = prf::input_count;
    static constexpr result_type min() { return prf::min(); }
    static constexpr result_type max() { return prf::max(); }
    static constexpr result_type default_seed = 20111115u;

    // constructors and seeding functions
    counter_based_engine() : counter_based_engine(default_seed) {}
    explicit counter_based_engine(result_type value);
    template<class Sseq> explicit counter_based_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

Pre-defined aliases of `counter_based_engine` are functionally identical to those in Section VI above.

```
// Philox engine with r rounds
template<int r>
using philox4x32_r = counter_based_engine<philox4x32_r_prf<r>, 4>;

// Philox engine with r rounds
template<int r>
using philox4x64_r = counter_based_engine<philox4x64_r_prf<r>, 4>;

// Philox engine with 10 rounds
using philox4x32 = counter_based_engine<philox4x32_prf, 4>;

// Philox engine with 10 rounds
using philox4x64 = counter_based_engine<philox4x64_prf, 4>;
```

## VIII. Further Work

- Range-based and SIMD-based APIs for random number engines have been proposed in other papers [12, 13]. If those proposals move forward, the APIs here must support them. We have a preliminary proof-of-concept that supports both range-based and SIMD-based APIs [14], but details remain to be worked out.
- Because of their small size and ease-of-construction, counter-based engines in general, and philox in particular are very well suited to the "independent streams" method of parallelization. A seed() method and corresponding constructor that gives the program direct control over all the elements of *K*, i.e., the fixed, non-incrementing state of the engine, would facilitate such usage.

## IX. Impact on the Standard

This is a library-only extension. It adds one or two new class templates, zero or one new concepts, and a small number of pre-defined template aliases.

## X. References

1. P1932R0 "Extension of the C++ random number generators": http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1932r0.pdf.
2. John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0
3. L'Ecuyer, Pierre & Simard, Richard. (2007). A Software Library in ANSI C for Empirical Testing of Random Number Generators. ACM Transactions on Mathematical Software - TOMS.
4. Manssen, Markus & Weigel, Martin & Hartmann, Alexander. (2012). Random number generators for massively parallel simulations on GPU. The European Physical Journal Special Topics. 210. 10.1140/epjst/e2012-01637-8.
5. Notes for Intel® Math Kernel Library (Intel® MKL) Vector Statistics : https://software.intel.com/en-us/mkl-vsnotes-philox4x32-10
6. Xu, Linlin & Ökten, Giray. (2014). High Performance Financial Simulation Using Randomized Quasi-Monte Carlo Methods. Quantitative Finance. 15. 10.1080/14697688.2015.1032549.
7. Wadden, Jack & Brunelle, Nathan & Wang, Ke & El-Hadedy, Mohamed & Robins, G. & Stan, Mircea & Skadron, Kevin. (2016). Generating efficient and high-quality pseudo-random behavior on Automata Processors. 622-629. 10.1109/ICCD.2016.7753349.
8. Random123 D. E. Shaw Research ("DESRES"): http://www.deshawresearch.com/resources_random123.html

9. N. Ferguson, S. Lucks, B. Schneier, B. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. http://www.schneier.com/skein.pdf, 2010.
10. J-P Aumasson and D. J. Bernstein. (2012). "SipHash: a fast short-input PRF", https://131002.net/siphash/
11. Y. Nir and A. Langley. (2018). "ChaCha20 and Poly1305 for IETF Protocols", https://tools.ietf.org/html/rfc8439
12. P1068R2 "Vector API for random number generation": http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1068r2.pdf.
13. P1932R3 "Vector API for random number generation": http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1068r3.pdf.
14. John Salmon's github: https://github.com/johnsalmon/cpp-counter-based-engine