*JFP* **25**, e6, 20 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S095679681500012X

# *Evaluation of splittable pseudo-random generators*[*]

HANS GEORG SCHAATHUN
*Aalesund University College, Pb. 1517, N-6025 Ålesund, Norway*
(*e-mail:* `georg@schaathun.net`)

**Abstract**

Pseudo-random number generation is a fundamental problem in computer programming. In the case of sequential processing the problem is very well researched, but parallel processing raises new problems whereof far too little is currently understood. Splittable pseudo-random generators (S-PRNG) have been proposed to meet the challenges of parallelism. While applicable to any programming paradigm, they are designed to be particularly suitable for pure functional programming. In this paper, we review and evaluate known constructions of such generators, and we identify flaws in several large classes of generators, including Lehmer trees, the implementation in Haskell's standard library, leapfrog, and subsequencing (substreaming).

## 1 Introduction

Pseudo-random number generation is one of the fundamental problems of computer sciences. Its history from computer infancy onward is documented, for instance, in Donald Knuth's classic book (1998). Random numbers are essential for many applications, including simulations, cryptography, random sampling, and gaming. However, true random values have two drawbacks. First, the entropy available is limited, and many applied problems would be intractable if limited by the availability of physical entropy. Secondly, many applications require repeatability, i.e. we want to be able to regenerate the same pseudo-random sequence in a later experiment. A pseudo-random number generator (PRNG) solves these two problems.

A classic PRNG is inherently sequential, and this is a major drawback in some applications. In a parallel program, each thread needs its own pseudo-random sequence, where each sequence is statistically independent of all the others. If new threads are created dynamically, new PRNGs must be seeded with the spawning thread's PRNG as the only source of entropy.

In a pure functional programming paradigm, the problem is even more fundamental. Without global state, the PRNG must be passed as an argument to every function requiring entropy. The bookkeeping of doing this in practice can be overcome by using monads. Unfortunately, such use of monads would still force

[*] The research was partially funded by Regionalt Forskingsfond Midt-Norge through the project *Dynamic Resource Allocation with Maritime Application* (DRAMA), grant no. ES504913.

an execution sequence in order to ensure deterministic behaviour. This sequencing prevents parallel execution on multi-core systems and potentially also some compiler optimisation.

Burton and Page (1992) introduced the concept of splittable PRNGs, which has been adopted by the standard library of Haskell. A function, split, is provided to return two generators $(g_1, g_2)$ from a single generator $g$. Thus, the calling function can split its own generator $g$ and pass $g_2$ to another function and keep $g_1$ for subsequent operations. The splittable PRNG does not create any dependency between evaluations, preserving referential transparency and leaving the compiler free to parallelise calculations. The split operation of Haskell was recently shown to be unsound by Claessen and Pałka (2013a) by providing an example of software failure in practice. They also proposed a splittable PRNG based on cryptographic hashing. Apart from Claessen and Pałka's work, very little work on splittable PRNGs can be found in the literature.

In this paper, we review and assess known constructions of S-PRNGs, finding flaws in most of them. In particular, we prove that there is an affine dependency in splittable PRNGs based on the classic Lehmer algorithm including the implementation in Haskell's standard library. We start with a literature review in Section 2, before we define mathematical concepts in Section 3. Section 4 reviews known constructions, and Section 5 evaluates them. Finally, we conclude in Section 6.

## 2 Historic overview and related research

Pseudo-random number generation for parallel processing has been studied since the 1980s. The idea of a splittable PRNG can be traced back to Warnock (1983), who discussed the simulation of particle physics. He suggested using three distinct linear recurrences. The first recurrence would be used to generate a seed for each of the particles at the start of simulation, thus associating with each particle its own generator. The second recurrence would be used to control the pseudo-random behaviour of an individual particle, using and updating the seed associated with this particle. The third recurrence would be used to generate a new seed to be associated with new particles being spawned. In this case, the spawning particle's seed would be used and not updated. Thus, the spawning of a new particle would not change the behaviour of the parent, since no number from the second recurrence is expended.

Warnock's generator can be seen to generate a tree, rather than a sequence, of pseudo-random numbers, and this came to be known as Monte Carlo trees. Frederickson *et al.* (1984) formalised the construction based on Lehmer's algorithm, under the notion of *Lehmer trees*. Several authors have studied methods to assess the randomness of Monte Carlo trees based on linear congruential generators (Halton, 1989; Percus & Kalos, 1989), and it is well known that the parameters used for the original Lehmer trees give poor randomness (Koniges & Leith, 1989; Eddy, 1990; Wu & Huang, 2006). Yet, there does not seem to have been any general results which would reject Lehmer trees or other linear congruential trees altogether. The Haskell library documentation only concludes that there has not been sufficient research on the matter (Hackage, 2011).

In the sequential case, linear congruential generators are extremely popular in the literature. Their statistical weaknesses have long been well known (Marsaglia, 1968; Krawczyk, 1992). In spite of the weaknesses, linear and combined congruential generators are satisfactory for many applications, and they dominate the reference literature in areas such as computation statistics (L'Ecuyer, 2012).

Whereas Warnock had been motivated by a practical simulation problem, aiming for data-parallel application of pseudo-random generators, Burton and Page (1992) discussed specifically the problem of pseudo-random generators in parallel, pure functional programs. They described the API of splittable PRNGs, including the split method that has been adopted by Haskell's System.Random library. Essentially, this split method is responsible for the branching in a Monte Carlo tree.

In spite of several useful applications of splittable generators and Monte Carlo trees, most of the literature has focused on parallel instantiation of sequential generators for multiprocessors systems. This topic can also be traced back to the 1980s, e.g. (Koniges & Leith, 1989; Eddy, 1990), and has also received recent attention (Salmon *et al.*, 2011). Even papers on Monte Carlo trees sometimes seem to have this simpler problem in mind (e.g. Frederickson *et al.* (1984)), using (say) the left-hand generator to branch and create a generator state per process. The right-hand generator is then used to generate the sequence for each of the processors. A very popular approach to parallelising an otherwise sequential PRNG is to split the output into subsequences or substreams. Several authors have studied different approaches to generate such subsequences (Matteis & Pagnutti, 1990; Burton & Page, 1992). Under the notion of *streams* and *substreams*, L'Ecuyer *et al.* (2002) discussed an object-oriented implementation to support extraction of parallel subsequences from a PRNG with long period. Wu and Huang (2006) discussed parameter selection for parallel linear congruential generators, and concluded that great care must be taken in the design.

Randomness testing has been studied by many authors. A classic, but still frequently cited reference is Knuth (1998). The TestU01 suite (L'Ecuyer & Simard, 2007) and Dieharder (Brown, 2015) are comprehensive software tools covering large ranges of tests. Several authors (Cuccaro *et al.*, 1995; Salmon *et al.*, 2011) have discussed randomness testing of parallel pseudo-random generators. Cuccaro *et al.* (1995) additionally discussed repeatability testing of splittable generators.

Matsumoto and Nishimura (1998) discussed dynamic creation of new pseudo-random generators by embedding an ID into the characteristic polynomial of the PRNG. The ID could for instance be a thread or process ID, which would give each thread or process different pseudo-random numbers. Matsumoto *et al.* (2007) discussed common defects of sequential pseudo-random generators and introduced the concepts of *nearly affine dependence* and *difference collisions*.

A very recent approach uses a hash function on the path from the root to the current node in the Monte Carlo tree. A convincing proposal (Claessen & Pałka, 2013a) simply encodes this path as a binary string and applies a cryptographic hash function to generate pseudo-random output. This solution is related to Leiserson *et al.* (2012) construction of a deterministic PRNG for dynamic threading (dthreading) architectures. Leiserson *et al.* also did some statistical testing of their proposed PRNG.

## 3 Definitions

### *3.1 Sequential number generation*

We view a PRNG as a state machine with a state space $\mathscr{S}$. The state transition $\mathsf{next}'(g) = (r, g')$ gives a new generator (state) $g'$ and a pseudo-random output $r$ from some range $\mathscr{R}$. More formally, we can define it as follows.

*Definition 1*
A PRNG is a tuple $G = (\mathscr{S}, \mathsf{next}, \mathscr{R}, \mathsf{ext})$, where $\mathscr{S}$ and $\mathscr{R}$ are finite sets and $\mathsf{next} : \mathscr{S} \to \mathscr{S}$ and $\mathsf{ext} : \mathscr{S} \to \mathscr{R}$ are functions. We write $\mathsf{next}'\, g = (\mathsf{ext} \circ \mathsf{next}\, g, \mathsf{next}\, g)$, where $\circ$ denotes function composition and binds more tightly than function application.

The definition of $\mathsf{next}'$ has been chosen to match most known implementations, and the Haskell standard library in particular, which calculate the new state first and then use it to calculate the output. One may argue that it would be better to use $\mathsf{next}''\, g = (\mathsf{ext}\, g, \mathsf{next}\, g)$.

The choice of range $\mathscr{R}$ matters little, as various well-known techniques exist to map a random value from one range and probability distribution into another. The design of such techniques is a problem separate from the design of good PRNGs. Most known PRNGs produce uniformly distributed integers in some range $(\mathscr{R}_{\min}, \mathscr{R}_{\max})$ as raw output, usually with $\mathscr{R}_{\min} = 0$ or $\mathscr{R}_{\min} = 1$. Generators which produce floating point values typically take the integer output $r$ and divide by $\mathscr{R}_{\max}$ or $\mathscr{R}_{\max} + 1$. Thus, a random value in the interval $(0, 1)$ is produced. The interval may be open or closed at either end.

A PRNG is instantiated by providing an initial state $g_0$, which is called the *seed*. A pair $(G, g)$ of a PRNG $G$ and a state $g \in \mathscr{S}$ will be called an *instance* of the PRNG. The instance defines the *pseudo-random sequence* $[x_i]_{i=1,2,\ldots}$ via the recurrence $(x_i, g_i) = \mathsf{next}'\, g_{i-1}$ and $g_0 = g$.

The period of a pseudo-random sequence $[x_i]$ is a well-known evaluation heuristic. It is the smallest number $t > 0$ such that $x_i = x_{i+t}$ for all $i \geqslant k$ for some $k$. All pseudo-random generators with finite state space give sequences with a finite period, underlining the fact that they are deterministic. Generally, the period must be sufficiently large to ensure that only a fraction thereof is ever used in a given application. The period of an instance $(G, g)$ of a PRNG is the period of the pseudo-random sequence it defines. Observe that the period is a property of the instance and not of the PRNG. Different instances of the same PRNG may have different period. We can define the *minimal period* of the PRNG to be the smallest period of any instance thereof.

A PRNG can be viewed as a directed graph where the state space $\mathscr{S}$ forms the set of nodes. Each node $s \in \mathscr{S}$ has out-degree one, with an edge $(s, \mathsf{next}\, s)$. We will use the graph view to illustrate key properties of splittable generators later.

The purpose of a PRNG is to generate number sequences which *appear* as if they were random. In cryptology, the common requirement is that no polynomial time algorithm exists able to distinguish between the PRNG output and a truly random sequence with probability significantly better than 50% (Menezes *et al.*, 1997). In simulation, a weaker requirement is common, where it suffices that the PRNG pass a battery of selected randomness tests (Knuth, 1998).

### *3.2 Parallel number generation*

A *parallel* (P-PRNG) is designed to support multiple parallel, independent instances of the generator. In other words, a P-PRNG can be instantiated to give a list $[(G^{(1)}, g^{(1)}), (G^{(2)}, g^{(2)}), \ldots, (G^{(N)}, g^{(N)})]$ of generator instances. To be useful, the pseudo-random sequences generated by the different constituent PRNG instances have to be statistically independent, in addition to each one being pseudo-random in its own right.

The typical application of a P-PRNG is to furnish each thread in a parallel program with its own generator, and the P-PRNG is suitable when the number of threads is known *a priori*. In this case, a P-PRNG can be instantiated and the constituent PRNG instances distributed across the threads. In contrast, the P-PRNG does not directly cater for new threads being spawned dynamically. There is no general way to find a new and unused PRNG instance at an arbitrary point in the program without using global state.

The PRNGs $G^{(i)}$ may or may not be distinct (Salmon *et al.*, 2011). In a multi-stream approach, a family of distinct PRNGs $G^{(i)}$ is required, typically by varying some parameter in the algorithm. For instance, in Lehmer's algorithm the multiplicative coefficient can be varied (Mascagni, 1998). The substream approach uses a single PRNG $G = G^{(i)}$ for all $i$ and varies only the state $g^{(i)}$. This means that the pseudo-random sequence is divided between the instances by choosing different starting points (seeds) $g^{(i)}$. The elements of a substreaming P-PRNG instance can be viewed as virtual generators (L'Ecuyer *et al.*, 2002), as the same real generator underlies every constituent instance.

### *3.3 Splittable number generators*

A splittable (S-PRNG) would provide an additional function:

$$\mathsf{split} : \mathscr{S} \rightarrow \mathscr{S} \times \mathscr{S}.$$

A typical application of split is when a subroutine is called, requiring a pseudo-random source. A call to split returns two generator instances; one of which can be fed to the callee, and one which can be used by the caller for further randomised behaviour. Thus, the callee does not have to return a modified state of the generator, and the caller does not have to process such a return value. Furthermore, as long as there are no other dependencies, subsequent operations of the caller and callee may be parallelised with great flexibility. Let us formalise the definition as follows.

*Definition 2*
A S-PRNG is a tuple $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$, where $\mathscr{S}$ and $\mathscr{R}$ are finite sets, $\mathsf{split}_L, \mathsf{split}_R, \mathsf{next}$ are functions $\mathscr{S} \to \mathscr{S}$, and $\mathsf{ext}$ is a function $\mathscr{S} \to \mathscr{R}$. We write $\mathsf{split}\, g = (\mathsf{split}_L\, g, \mathsf{split}_R\, g)$.

Note that the functions $\mathsf{split}_L, \mathsf{split}_R, \mathsf{next}$ are not necessarily bijective, just as $\mathsf{next}$ does not have to be bijective in a sequential PRNG. The definition of $\mathsf{split}$ can be extended to instances, so that

$$\mathsf{split}(G, g) = ((G, \mathsf{split}_L\, g), (G, \mathsf{split}_R\, g)).$$

An S-PRNG is more general than a P-PRNG. The P-PRNG provides only a single pool of generator instances, whereas the S-PRNG allows new parallel instances to be created whenever one instance is available. Given an instance $(G, g_0)$ of an S-PRNG, we can instantiate a P-PRNG via the recurrence $g_i = \mathsf{split}_R\, g_{i-1}$ and use $(G, \mathsf{split}_L\, g_i)$ as constituent generator instances.

Like other PRNGs, the S-PRNG is a state machine, which in turn can be viewed as a directed graph. For the S-PRNG, the nodes have out-degree at most three, with edges defined by $\mathsf{next}$, $\mathsf{split}_L$, and $\mathsf{split}_R$ respectively. Smaller out-degree is caused by two or three of the maps (edges) coinciding on a particular node. A collision between the two split operations should only occur for a negligible fraction of the state space, to avoid statistical deficiencies. In contrast, some constructions use $\mathsf{next} = \mathsf{split}_L$ or $\mathsf{next} = \mathsf{split}_R$, giving every node out-degree at most two, and this causes no problem since $\mathsf{split}$ and $\mathsf{next}$ are never both applied to the same node. Paths define strings over the alphabet $\{\mathsf{next}, \mathsf{split}_L, \mathsf{split}_R\}$, and we call them operations on the S-PRNG.

*Definition 3*
Consider an S-PRNG $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$. An *operation* of length $m$ on $G$ is a string $s = [f_1, f_2, \ldots, f_m]$ of $m$ tokens from the alphabet $\{\mathsf{next}, \mathsf{split}_L, \mathsf{split}_R\}$. We write $L(s) = m$ for the length.

By function composition, any operation $s = [f_1, f_2, \ldots, f_{n-1}, f_n]$ defines a function on $\mathscr{S}$. By abuse of notation, we will use the same notation for the function and the operation, and write

$$s = f_n \circ f_{n-1} \circ \cdots \circ f_2 \circ f_1.$$

Note that the set of all functions $\mathscr{S} \to \mathscr{S}$ is finite, whereas the operations, defined as strings of arbitrary length, make an infinite set.

The periodicity of pseudo-random sequences corresponds to cycles in the graph defined by the (S-)PRNG.

*Definition 4*
Consider an S-PRNG $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$. A cycle of $g \in \mathscr{S}$ is an operation $s$ such that $s(g) = g$. A (generic) cycle of $G$ is an operation $s$ which is a cycle for every $g \in \mathscr{S}$.

Where the sequential generator allows only a single path between two given states $g_1$ and $g_2$, the S-PRNG may have multiple paths between any pair of states. If there are two short paths from $g_1$ to $g_2$, there is a significant risk that $g_2$ be used twice once $g_1$ has been used, and this would lead to significant bias. Therefore, we introduce a couple of new concepts to be able to discuss such paths in depth.

*Definition 5*
Let $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$ be an S-PRNG and $g \in \mathscr{S}$ a state. A bad pair of operations of length $m$ for $g$ is a pair of operations $(s_1, s_2)$ where

1. $s_1(g) = s_2(g)$.
2. The first element where $s_1$ and $s_2$ differ has to be a split (neither operation may have next).
3. $m = \max(L(s_1), L(s_2))$, where $L$ denotes the length of an operation.

The second restriction means that both operations can be applied to the same state in a valid program, where next should never be called on the same state as split. The condition permits a common prefix $s_p$ such that $s_1 = s_1' \circ s_p$ and $s_2 = s_2' \circ s_p$. It requires, however, that if $s_p$ is the longest common prefix, then $s_1'$ has to start with $\mathsf{split}_L$ and $s_2'$ with $\mathsf{split}_R$ or vice versa. Note that if $(s_1, s_2)$ is a bad pair of operations for $g$, then $(s_1', s_2')$ is a bad pair of operations for $s_p(g)$ and vice versa.

*Definition 6*
Let $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$ be an S-PRNG. A (consistently) bad pair of operations for $G$ is a pair $(s_1, s_2)$ which is a bad pair of operations for every $g \in \mathscr{S}$.

*Remark 1*
If a consistently bad pair of operations of length $m$ exists for an S-PRNG $G$, then there is a program which always repeats a state, regardless of the initial seed, making no more than $2m$ applications of split or next on $G$.

Where a sequential PRNG defines a (periodic and infinite) sequence of states $[g_1, g_2, \ldots]$, the S-PRNG defines a *Monte Carlo tree*. Given a root $g_0$, an infinite, ternary tree is obtained where each node is labelled with a state $g \in \mathscr{S}$ and has three children labelled $\mathsf{split}_L\, g$, $\mathsf{split}_R\, g$, and $\mathsf{next}\, g$ respectively. A node can be identified by the root $g_0$ and the path therefrom, that is an operation on the S-PRNG. In an application, one would not call split and next on the same state (Claessen & Pałka, 2013a), so the states actually traversed will form a binary subtree.

## 4 Known proposals for splittable PRNG

### *4.1 Lehmer trees*

The original Lehmer algorithm defines a sequence of pseudo-random numbers via the recurrence

$$s_i = a \cdot s_{i-1} + c \mod m. \tag{1}$$

The output is equal to the new state, that is $\mathsf{ext}\, s = s$.

Based on this algorithm, Frederickson *et al.* (1984) introduced Lehmer trees. We need a left and right state transition function with different coefficients, defined as follows:

$$\mathsf{split}_R(s) = a_R \cdot s + c_R \mod m, \tag{2}$$
$$\mathsf{split}_L(s) = a_L \cdot s + c_L \mod m. \tag{3}$$

### *4.2 Haskell's System.Random*

The Haskell library implements the *Portable Combined Generator* of L'Ecuyer (1988) for 32-bit computers with an additional split operation. It is a Multiple Recursive Generator (MRG) combining two LCGs. The period is roughly $2.30584 \cdot 10^{18}$, with an output range of more than 30 bits. The combined generator is designed to be at least as statistically robust as the *Minimal Standard Random Number Generator* described by Park and Miller (1988) and Carta (1990). more than 30 bits. The Combined Generator is composed of two constituent random generators, each using Lehmer's algorithm with $c = 0$ and the following values for $a$ and $m$:

$$a_1 = 40{,}014 \quad m_1 = 2{,}147{,}483{,}563 \tag{4}$$
$$a_2 = 40{,}692 \quad m_2 = 2{,}147{,}483{,}399. \tag{5}$$

This gives $\mathscr{S} = \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2}$ as the state space. The state transition function will advance each constituent generator to the next state $s_i'$, i.e. we write the function as

$$\begin{aligned} &\mathsf{next} : (s_1, s_2) \mapsto (s_1', s_2'), \\ &\text{where } s_i' = a_i s_i \mod m_i. \end{aligned} \tag{6}$$

The random integer output $z$ is obtained as the difference between the two substates (constituent outputs), adjusted to fit the desired range, that is

$$\mathsf{ext}(s_1, s_2) = \begin{cases} (s_1 - s_2) + m_1 - 1 & \text{if } (s_1 - s_2) < 1 \\ (s_1 - s_2) & \text{otherwise.} \end{cases} \tag{7}$$

Thus, the range is $\mathscr{R} = \{1, 2, \ldots, m_1 - 1\}$.

No split function was defined by L'Ecuyer (1988), but Haskell implements the following definition.

$$\begin{aligned} \mathsf{split} :&(s_1, s_2) \mapsto ((s_1', s_2''), (s_1'', s_1')), \\ s_1'' &= \begin{cases} 1 & \text{if } s_1 = m_1 - 1 \\ s_1 + 1 & \text{otherwise} \end{cases} \\ s_2'' &= \begin{cases} m_2 - 1 & \text{if } s_2 = 1 \\ s_2 - 1 & \text{otherwise} \end{cases} \\ (s_1', s_2') &= \mathsf{next}(s_1, s_2). \end{aligned}$$

This essentially turns each substate into a Lehmer tree, except for the correction to avoid zero states, something which is necessitated by the zero additive constant in

Equation (6). There is no statistical foundation for the split function defined above, and the documentation (Hackage, 2011) says:

> *Until more is known about implementations of 'split', all we require is that 'split' deliver generators that are (a) not identical and (b) independently robust (...)*

### *4.3 Substreaming*

Substreaming (also known as subsequencing) of congruential generators is a common approach to construct a P-PRNG (Wu & Huang, 2006) and has been suggested for S-PRNG as well (Burton & Page, 1992). We will need the following auxiliary definition.

*Definition 7*
Let $A$ be an arbitrary set. For any function $f : A \to A$, we write $f^{(1)} = f$ and define recursively $f^{(n)} = f \circ f^{(n-1)}$ for $n > 1$,

A pseudo-random sequence $[x_1, x_2, \dots, x_n]$ can be subsequenced in two different ways. Regularly spaced segments (also known as substreams) means that we split the sequence into blocks $[x_1, x_2, \dots, x_{\lceil n/2 \rceil}]$ and $[x_{\lceil n/2 \rceil+1}, x_{\lceil n/2 \rceil+2}, \dots, x_n]$. Depending on the underlying PRNG, it may or may not be necessary to generate the entire sequence explicitly to make the split.

*Definition 8*
For any PRNG $G = (\mathscr{S}, f, \mathscr{R}, \mathsf{ext})$, we define a corresponding S-PRNG by regularly spaced segments as $G' = (\mathscr{S}, f, \mathsf{split}_L, \mathsf{split}_R, \mathscr{R}, \mathsf{ext})$ where

$$\mathsf{split}\, s = (s, f^{(\lceil n/2 \rceil)}(s)).$$

where $n$ is the period of the PRNG.

The leapfrog construction takes odd indices for one subsequence and even ones for the other, to get $[x_1, x_3, \dots]$ and $[x_2, x_4, \dots]$. The easiest way to define leapfrog is to ignore the definition of S-PRNG and define the $\mathsf{split}$-operation in terms of generator instances instead of generator states. Any S-PRNG can be fit into this framework by setting $G_1 = G_2 = G$ and $(g_1, g_2) = \mathsf{split}\, g$. To define leapfrog, write $G = (\mathscr{S}, \mathsf{next}, \mathscr{R}, \mathsf{ext})$, and let $\mathsf{split}(G, g) = ((G_1, g_1), (G_2, g_2))$ be given by

$$g_1 = \mathsf{next}\, g \quad \text{and} \quad g_2 = \mathsf{next} \circ \mathsf{next}\, g,$$
$$G_1 = G_2 = (\mathscr{S}, \mathsf{next} \circ \mathsf{next}, \mathscr{R}, \mathsf{ext}).$$

There is an S-PRNG construction which is equivalent to the above. We observe that the $\mathsf{next}$-function is replaced whenever an instance is split. Thus, we need to integrate this function into the state.

*Definition 9* (*Leapfrog*)
For any PRNG $G = (\mathscr{S}, f, \mathscr{R}, \mathsf{ext})$, we define a corresponding S-PRNG by leapfrog as $G' = (\mathscr{S}', \mathsf{next}, \mathsf{split}_L, \mathsf{split}_R, \mathscr{R}, \mathsf{ext})$ where

$$\begin{aligned}\mathscr{S}' &= \mathscr{S} \times \{f^{(i)} : i = 1, 2, 3, \dots\},\\ \mathsf{next}(g, f) &= (f(g), f),\\ \mathsf{split}_L(g, f) &= (f(g), f \circ f),\\ \mathsf{split}_R(g, f) &= (f \circ f(g), f \circ f).\end{aligned}$$

Note that $\mathscr{S}'$ is finite. The second element of the pair is a function $\mathscr{S} \to \mathscr{S}$, and since $\mathscr{S}$ is finite, there is a finite number of such functions, and hence the $f^{(i)}$ cannot all be distinct.

## 4.4 Random jumps

A pseudo-random sequence can be split by using the sequence itself to choose random positions in the sequence (Burton & Page, 1992). Given a PRNG, consider the sequence of states $[g_0, g_1, \dots]$ where $g_i = \mathsf{next}\, g_{i-1}$ and $g_0$ is the initial state, and let $\mathsf{rand}\, g = g_{\mathsf{ext}\, g}$. The split operator can be defined as

$$\mathsf{split}\, g = (\mathsf{rand}\, g, \mathsf{next}\, g). \tag{8}$$

We have not seen any concrete proposals for splittable generators based on this generic construction, but it is easy enough to implement it with the Haskell's Portable Combined Generator.

It is tempting to adapt the random jump construction for combined generators, applying (8) differently to each constituent generator. For instance, we can use the following for multiple congruential generators:

$$\begin{aligned}\mathsf{split} :&(g_1, g_2) \mapsto ((g_1', g_2''), (g_1'', g_1')), \quad \text{where} \qquad (9)\\ (g_1', g_2') &= \mathsf{next}(g_1, g_2)\\ g_1'' &= a_1^{g_1} \cdot s_1 \mod m_1\\ g_2'' &= a_2^{g_2} \cdot s_2 \mod m_2,\end{aligned}$$

where $s_i$ is the initial state (seed) of constituent generator $i$. We note that the exponentiation in the definitions of $g_i''$ will make $\mathsf{split}$ computationally much more expensive than $\mathsf{next}$; yet with square-and-multiply the cost is likely to be acceptable.

## 4.5 The Claessen–Pałka generator

Claessen–Pałka (2013a) introduced an S-PRNG based on cryptographic hashing. In essence, it is a generalisation of counter mode, which allows any any block cipher to be used as a sequential PRNG.

*Definition 10* (*Counter mode*)

Let $e_k : \mathbb{Z}_{2^m} \to \mathbb{Z}_{2^m}$ be an encryption function of some block cipher. The elements of $\mathbb{Z}_{2^m}$ are viewed dually as $m$-bit strings and integers modulo $2^m$. Counter mode defines the PRNG $G = (\mathbb{Z}_{2^m}, \mathsf{next}, \mathbb{Z}_{2^m}, \mathsf{ext})$, where

$$\begin{aligned} \mathsf{next}\, s &= s + 1 \mod 2^m, \\ \mathsf{ext}\, s &= e_k(s). \end{aligned}$$

Instead of the integer state where $\mathsf{next}$ adds one, we can use a binary string and append a 1. Replacing the encryption $e_k$ with a hash function $h_k$, strings of arbitrary length can be handled. The split operation can then be defined by appending 0 in one branch and 1 in the other, as follows.

*Definition 11*

Given some hash function $h_k$, we define an S-PRNG where the state space is the set $\{0,1\}^*$ of all binary strings, and

$$\begin{aligned} \mathsf{ext}\, \mathbf{x} &= h_k(\mathbf{x}), \\ \mathsf{split}_L\, \mathbf{x} &= \mathbf{x}||0, \\ \mathsf{split}_R\, \mathbf{x} &= \mathbf{x}||1, \\ \mathsf{next}\, \mathbf{x} &= \mathsf{split}_R\, \mathbf{x}, \end{aligned}$$

where $||$ denotes concatenation.

Using the Merkle–Damgård construction for $h_k$, it is not necessary to store the entire string $\mathbf{x}$ for the state. The string is split into blocks of fixed length, and it suffices to store the last (possibly incomplete) block alongside the hash of all preceding blocks. Thus the state space is made finite, and the time and memory complexities become constant.

The final construction of Claessen and Pałka (2013a) includes a couple of variations and tricks to optimise the implementation. We omit details which are not significant for the mathematical analysis. They provide the tf-random package (Claessen & Pałka, 2013b) for Haskell, using ThreeFish as the block cipher underlying the Merkle–Damgård hash construction.

Clearly, other hash functions can be used instead of ThreeFish and Merkle–Damgård. Leiserson *et al.* (2012) discussed a couple of alternative hash functions for a similar PRNG construction, and their considerations are valid also for the Claessen–Pałka construction.

There is an inverse construction as well. Given an S-PRNG, we can construct a hash function as follows. Let the secret key be the seed $g$. To hash a binary string $\mathbf{x}$, we map it into an operation $s$ by $1 \mapsto \mathsf{split}_R$ and $0 \mapsto \mathsf{split}_L$. The hash value of $\mathbf{x}$ is $\mathsf{ext} \circ s(g)$. Claessen and Pałka (2013a) pointed out this equivalence of constructions, but did not formally prove any equivalence between the anti-collision properties of hash functions and randomness of an S-PRNG.

## 5 Assessment of the known constructions

### *5.1 Risk of repeating states in general*

With a sequential PRNG of period $N$, one may expect to safely use $O(N)$ random numbers. There are several arguments to say that an S-PRNG can never achieve this linearity. If we want an absolute guarantee against repeated states, we can only use $O(\log N)$ random numbers in general, as the following proposition shows.

*Proposition 1*
Let $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$ be an S-PRNG, where $N = \#\mathscr{S} \geqslant 2$. For any $g \in \mathscr{S}$, there is a bad pair of operations for $g$, with length at most $m = \lceil \log_2 N \rceil$.

*Proof*
Consider a complete finite subtree $T$ of the Monte Carlo tree, comprising the root $g$ at level 0 and all children generated by split down to level $m$ inclusive. This tree has $2^{m+1} - 1$ nodes. Since $\#S = N \leqslant 2^m < 2^{m+1} - 1$, there must be a repeated state $g'$ in $T$. The paths to the two nodes labelled $g'$ form a bad pair of operations $(s_1, s_2)$. Since both operations are paths within a tree of depth $m$, the length of the pair is at most $m$. □

Note that this proposition does not imply that there is a bad pair of operations for $G$. Different pairs may be bad for different states. In many applications, it may be possible to control the negative effect of bad pairs of operations by rerunning the application with different seeds. Unless the same pair is bad for a large proportion of the state space, it is unlikely to be repeated.

Another bound, saying that we cannot use more than $O(\sqrt{N})$ numbers from an S-PRNG in general, stems from the birthday paradox, and it has previously been pointed out by Wu and Huang (2006) in the context of random jumps. If elements are drawn randomly from a set of $N$ elements without replacement, the expected number $n$ of elements drawn before a repetition is found, is $O(\sqrt{N})$ (Klamkin & Newman, 1967). Both random jumps and the hash-based S-PRNG have a split function where $\mathsf{split}_L$ aims to jump to a random location in the sequence generated by $\mathsf{split}_R$ or $\mathsf{next}$ (and $\mathsf{split}_R = \mathsf{next}$). If the left split is truly random, then the birthday paradox clearly applies and implies that a collision is expected before $O(\sqrt{N})$ splits where $N$ is the size of the state space. Admittedly, we do not have true randomness and thus it may not be entirely clear what really happens. However, it is fair to say that the only cure against the birthday paradox in an S-PRNG is poor randomness.

### *5.2 Substreaming*

Substreaming is an important tool for parallel pseudo-random number generation, but it is inadequate as a splittable PRNG. If we consider the pseudo-random sequence to have finite length $N$, it is obvious that $\log_2 N$ consecutive splits leave us with segments of length 1. Even the Mersenne Twister with $N \approx 2^{65,000}$ would run out of random numbers in a medium sized application. If we view the sequence as infinite and periodic, we risk repeated states. This can be phrased formally as follows.

*Proposition 2*
Let $G$ be the S-PRNG obtained by regularly spaced segmenting of some PRNG instance of period $N$. Then, $\mathsf{next} \circ \mathsf{split}_R^{(m-1)}$ is a generic cycle of length $m = \lfloor \log_2 N \rfloor + 1$ for $G$.

*Proof*
Let $[g_1, g_2, \ldots, g_N]$ be the sequence of states traversed by the underlying PRNG instance. Observe that $\mathsf{split}_R^{(m-1)}(g_1) = g_N$, and hence $\mathsf{next} \circ \mathsf{split}_R^{(m-1)}(g_1) = g_1$ as required. Since the sequence of states is periodic, we get the same result regardless of which state is chosen as $g_1$. □

## 5.3 Leapfrog

Leapfrog effectively changes the $\mathsf{next}$ operation when an instance is split, and this makes it a little harder to analyse. First observe that a state (in general) may be *repeated* or *unreachable*. A repeated state $g$ has the property that there is a $t \geqslant 1$, such that $\mathsf{next}^{(t)}\, g = g$ and forms an orbit $\mathsf{orb}(g) = \{\mathsf{next}^{(i)}\, g | i = 1, 2, \ldots, t\}$. When the PRNG is seeded by $g$, $\mathsf{orb}(g)$ is the set of states which may be used in the program. An unreachable state has no such $t$ and is not in any orbit, even though it must be part of a sequence ending up in an orbit. Observe that the orbits may have different orders, and hence not every sequence produced by the PRNG has the same period.

Studying a single orbit at a time, it is possible to find consistently bad pairs of operations for an S-PRNG based on leapfrog. Given a PRNG $G = (\mathscr{S}, \mathsf{next}, \mathscr{R}, \mathsf{ext})$, we write $\mathscr{S}_g = \mathsf{orb}(g)$ and define $G_g = (\mathscr{S}_g, \mathsf{next}, \mathscr{R}, \mathsf{ext})$. Observe that $\mathsf{next}$ is a bijection on $\mathscr{S}_g$.

We need to distinguish between odd and even period. Note that $\mathsf{next}$ generates a group $P$ acting on $\mathscr{S}_g$. If $P$ has even order $N = \#P$, then $\mathsf{next} \circ \mathsf{next}$ generates a subgroup of order $N/2$. Consequently, the generator instances created by $\mathsf{split}_R$ and $\mathsf{split}_L$ each use disjoint halves of the state space. In the extreme case where the period is a power of two, repeated splitting leads to generators with singleton state space, where no further splitting is possible. If $N$ is odd, then $\mathsf{next} \circ \mathsf{next}$ generates $P$ and $\mathsf{split}_R$ and $\mathsf{split}_L$ yield instances using the entire state space $\mathscr{S}_g$ in different orderings.

*Lemma 1*
Let $G_g$ be a PRNG with a single orbit of odd order $N$. Splitting $G_g$ using leapfrog gives an S-PRNG with a bad pair of operations of length at most $2\lceil \log_2 N \rceil - 1$.

*Proof*
Write $m = \lceil \log_2 N \rceil$. Consider the two operations

$$s_1 = \mathsf{split}_R^{(m-1)} \circ \mathsf{split}_L$$
$$s_2 = \mathsf{split}_L^{(m-1)} \circ \mathsf{split}_R,$$

and apply them to an arbitrary state $(g,f)$. We get

$$s_1(g,f) = (g_1,f_1) = (f^{(r_1)}(g), f^{(2^m)})$$
$$s_2(g,f) = (g_2,f_2) = (f^{(r_2)}(g), f^{(2^m)}).$$

If $g_1 = g_2$ then $(s_1,s_2)$ is a bad pair of operations as required, so consider the case where $g_1 \neq g_2$. Then $g_1,g_2$ are distinct elements of the cycle generated by $G_g$. Since the period $N$ is odd, there is an even number $1 \leqslant N_0 < N$ such that either $g_1 = f^{(N_0)}(g_2)$ or $g_2 = f^{(N_0)}(g_1)$. Because of symmetry, it is sufficient to consider the first case.

We will design an operation $s_2'$ such that

$$s_2'(g,f) = (g_1,f_1) = (f^{(N_0+r_2)}(g), f^{(2^m)}),$$

which would make $(s_1,s_2')$ a bad pair of operations. We will design $s_2'$ by inserting extra next-operations into $s_2$, so consider

$$s_2' = q_{m-1} \circ q_{m-2} \circ \cdots \circ q_1 \circ \mathsf{split}_R,$$

where $q_i$ is either $\mathsf{split}_L$ or $\mathsf{split}_L \circ \mathsf{next}$. Let $Q$ be the set of indices for which $q_i = \mathsf{split}_L \circ \mathsf{next}$. We get that

$$s_2'(g,f) = (g_2',f_2) = (f^{(r_2+N')}(g), f^{(2^m)}),$$

where

$$N' = \sum_{i \in Q} 2^i.$$

It is easy to see that $Q$ can be chosen such that $N' = N_0$, and thus we get a bad pair of operations $(s_1,s_2')$ of length at most $2m-1$. $\square$

*Proposition 3*
Let $G_g$ be a PRNG with a single orbit. Splitting $G_g$ using leapfrog gives an S-PRNG with a bad pair of operations of length at most $2\lceil \log_2 N \rceil - 1$ where $N$ is the period.

*Proof*
Write $\#\mathscr{S} = 2^k \cdot N'$ where $N'$ is odd. Consider the instance $(G,g)$ where $g$ has period $N$. Then the instance $(G, \mathsf{split}_L^{(k)} g)$ has period $N'$ and by Lemma 1, it has a bad pair of operations $(s_1,s_2')$. In the proof of the lemma, we note that $(s_1,s_2')$ depends only on the order $N'$ and not on the particular instance. Hence $(s_1 \circ \mathsf{split}_L^{(k)}, s_2 \circ \mathsf{split}_L^{(k)})$ is a bad pair of operations for $G_g$ of length at most $2\lceil \log_2 N \rceil - 1$. $\square$

One might think that a result on a single orbit is very limiting. However, the sum of the periods of all the orbits is upper bounded by the size of the state space. Hence, having many orbits would be at the expense of the period. Furthermore, if the PRNG has multiple orbits of orders $N_1, N_2, \ldots, N_n$, then the argument of the proofs of Lemma 1 and Proposition 3 can be used to construct a consistently bad pair of operations of length at most $2\lceil \log_2 \ell \rceil - 1$ where $\ell$ is the least common multiple of all the $N_i$. Thus, to ensure that any bad pair of operations is very long, the state space necessarily becomes very large compared to the minimal period.

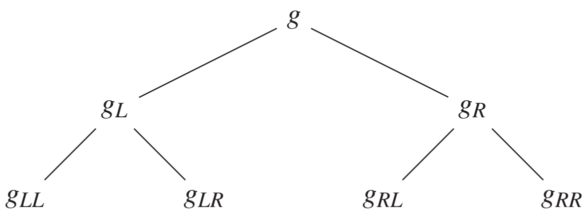

Fig. 1. Split generators for the quad-test.

### *5.4 Lehmer trees and the Haskell generator*

Consider the generators arising when we split twice, as shown in Figure 1. We shall see that the Lehmer tree gives a dependency between $g_{LR}$ and $g_{RL}$. The generator implemented in the Haskell standard library is a variation of Lehmer trees, and has the same problem. We consider the two generators $g_{LR}$ and $g_{RL}$ as well as their parents and grandparent.

*Theorem 1*

Let $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$ be a Lehmer tree. For any initial state $g$, there is a constant difference between the two generator states $g_{RL} = \mathsf{split}_L \circ \mathsf{split}_R\, g$ and $g_{LR} = \mathsf{split}_R \circ \mathsf{split}_L\, g$, as follows:

$$g_{RL} - g_{RL} \equiv (a_L - 1)c_R - (a_R - 1)c_L \pmod{m},$$

where $c_L$ and $c_R$ are the left and right additive constants and $a_L$ and $a_R$ are the corresponding multiplicative constants used by the Lehmer functions.

*Proof*

Consider the compositions

$$\mathsf{split}_L \circ \mathsf{split}_R\, g = a_L a_R \cdot g + a_L c_R + c_L \mod m \tag{10}$$

$$\mathsf{split}_R \circ \mathsf{split}_L\, g = a_R a_L \cdot g + a_R c_L + c_R \mod m, \tag{11}$$

and taking the difference, we get the congruence stated in the proposition. □

*Theorem 2*

Let $G = (\mathscr{S}, \mathsf{split}_L, \mathsf{split}_R, \mathsf{next}, \mathscr{R}, \mathsf{ext})$ be the Haskell S-PRNG as defined in Section 4.2. Consider a random state $g$, and write $g_{LR} = \mathsf{split}_L \circ \mathsf{split}_R\, g$ and $g_{RL} = \mathsf{split}_R \circ \mathsf{split}_L\, g$. Except with negligible probability $\leqslant 3 \cdot 2^{-30}$, the following congruences hold:

$$g_{RL}^{(1)} - g_{LR}^{(1)} \equiv a_1 - 1 \pmod{m_1}$$
$$g_{RL}^{(2)} - g_{LR}^{(2)} \equiv a_2 - 1 \pmod{m_2}.$$

where $g_*^{(1)}$ and $g_*^{(2)}$ denote the two substates of $g_*$.

*Proof*

We will prove that the theorem holds except when $g_*^{(1)} = 1$ or $g_*^{(2)} = m_2 - 1$ for some $g_* \in \{g, g_L, g_R\}$. Observe that when $g_* \in \mathscr{S}$ is drawn at random, we have $P(g_*^{(1)} = 1) = 1/m_1$ and $P(g_*^{(2)} = m_2 - 1) = 1/m_2$. Each probability is roughly $2^{-31}$, and the probability that one of the events occur for one of the states $g, g_L, g_R$ is less than $3 \cdot 2^{-30}$.

Ignoring these improbable cases, we write

$$\begin{aligned} g_{LR}^{(1)} &= a_1 g^{(1)} + 1 \mod m_1 \\ g_{LR}^{(2)} &= a_2(g^{(2)} - 1) \mod m_2 \\ g_{RL}^{(1)} &= a_1(g^{(1)} + 1) \mod m_1 \\ g_{RL}^{(2)} &= a_2 g^{(2)} - 1 \mod m_2, \end{aligned}$$

and by subtracting the first from the third equation, and second from the fourth, we get the two congruences stated in the theorem. □

Similar dependencies exist between the pseudo-random output generated by $g_{LR}$ and $g_{RL}$. Let $[g_i]$ and $[g_i']$ be the sequences of states generated recursively by next starting at $g_{LR}$ and $g_{RL}$ respectively. It is easy to see that

$$\begin{aligned} g_i^{(1)} - g_i'^{(1)} &\equiv a_1^i(a_1 - 1) \pmod{m_1} \\ g_i^{(2)} - g_i'^{(2)} &\equiv a_2^i(a_2 - 1) \pmod{m_2}. \end{aligned}$$

Observing the ext-function in (7), we note that there are only four possible values for $\mathsf{ext}\, g_i - \mathsf{ext}\, g_i'$.

## *5.5 Statistical testing*

The proofs of Theorems 1 and 2 are self-contained pieces of algebra, but they were not conceived as such. Our approach started with statistical testing, which was necessary to identify the offending combinations of states.

A comprehensive discussion of testing is beyond the scope of this paper. We have only tried a few simple tests sufficient to find flaws in most of the constructions studied. Our tests are based on the serial test (Knuth, 1998), which is well known for testing of sequential generators. It is the simplest and most straight forward among the tests for correlation between elements in a sequence.

*Definition 12* (*Serial test*)
An integer random sequence $X$ is chopped into disjoint $t$-tuples. Each integer is reduced to $b$ bits by masking out a selected block. Thus, we get a sequence $X'$ of words of $tb$ bits each. The *serial test* applies the $\chi^2$ test to test if the elements of $X'$ are uniformly distributed. Thus, the $\chi^2$ statistic is

$$\chi^2 = \frac{\sum \left( \frac{N}{2^{tb}} - h_i \right)}{N},$$

where $h_i$ is the histogram of the observed values and $N$ is the length of $X'$.

The masking of $b$ bits is necessary to reduce the sample space. If $b$ is too large, an enormous sample would be required to get reasonable significance. In our tests, we have used $(t, b) = (4, 2)$ and $(t, b) = (2, 4)$, each of which gives eight-bit blocks.

To generate the pseudo-random number sequence $X$, we generate a sequence of states $S = [g_1, g_2, \ldots]$ from a single seed $g_0'$ using the recurrence $(g_i', g_i) = \mathsf{split}\, g_{i-1}'$.

For each element $g_i$, we generate the tuple $(g_{LL}, g_{LR}, g_{RL}, g_{RR})$ as defined in the previous subsection. From each of the four generators, we take one random number $r_X = \mathsf{ext} \circ \mathsf{next}\, g_X$. The result is a list of integer four-tuples $\mathbf{r} = (r_{LL}, r_{LR}, r_{RL}, r_{RR})$. Juxtaposing the tuples into a single integer sequence $X$, we apply the serial test with $t = 4$ as described in Definition 12. We call this test the *quad-test*. A total of 16 tests are run. Each test considers a block of two adjacent bits, starting at bit no. $1, 3, 5, \ldots, 29$ and 30, counting from the least significant bit. The output range is 31 bits, so the 32nd bit is biased and must be ignored.

Running the quad-test on Haskell's standard generator, we consistently got a $p$-value of 0 to four decimal places. Running a second set of tests, considering only two out of four generators per tuple, the only offending pair was $(r_{LR}, r_{RL})$. This observation led to the proofs in the previous subsection.

The flaw in Haskell's generator is closely tied to the linearity of the maps used by split. We tried a variation replacing the map $s \mapsto s \pm 1$ in the $\mathsf{split}$ function, by a quadratic function $s \mapsto s^2$, and the resulting S-PRNG passed the tests.

## 5.6 Statistical evaluation of random jumps

We have evaluated both variations of random jumps defined in (8) and (9) using statistical testing. Both passed the quad-test described above, but the original version (8) fails the *split sequence test* which we describe below.

Starting with some seed $g'_0$, we define the following split sequences:

$$S_A = [g_1, g_2, \ldots], \quad \text{where} \begin{cases} (g_i, g'_i) = \mathsf{split}\, g'_{i-1}, & \text{if } i \text{ is odd} \\ (g'_i, g_i) = \mathsf{split}\, g'_{i-1}, & \text{if } i \text{ is even} \end{cases} \tag{12}$$

$$S_L = [g_1, g_2, \ldots], \quad \text{where } (g_i, g'_i) = \mathsf{split}\, g'_{i-1} \tag{13}$$

$$S_R = [g_1, g_2, \ldots], \quad \text{where } (g'_i, g_i) = \mathsf{split}\, g'_{i-1}. \tag{14}$$

As before, each sequence $S$ of generators gives rise to an integer sequence $X = [x_1, x_2, \ldots]$ where $x_i = \mathsf{ext} \circ \mathsf{next}\, g_i$, and this sequence is made subject to the serial test (Definition 12).

For each of the three sequences, we run eight tests with $(t, b) = (2, 4)$ and 16 sets with $(t, b) = (4, 2)$, using different bit blocks. Additionally, we ran the 16 quad-tests, for a test suite of 88 tests total. The actual test suite used included 135 tests, but the the remaining 47 tests gave no interesting information and we leave them undocumented. The entire test suite was run four times indepenently, each time with a sample size of $n = 25{,}000$ tuples ($25{,}000 \cdot t$ random numbers).

The original definition of random jump split (8) was made subject to the tests as described. A number of the tests failed with a $p$-value of 0 to four decimal places. In the first run, this happened for the 16 tests with $(t, b) = (4, 2)$ on $S_L$. In the second run, it happened to all 24 tests on $S_A$ as well as 14 out of 16 tests with $(t, b) = (4, 2)$ on $S_L$ (the remaining two tests had $p$-value $< 0.0002$). In the last two runs, it happened to the 16 tests with $(t, b) = (4, 2)$ on $S_A$. The third run also had a large number of tests on $S_L$ failing with $p$-value below 5%. It is clear that the original definition of Random Jumps is unsound when applied to the Portable Combined Generator.

Changing the split function to Equation (9), the generator passes all of the tests that we have proposed. Even so, there is little reason to trust this S-PRNG. The construction is only an *ad hoc* variation of the original random jump construction which failed, and it may very well fail similar tests on other branches in the Monte Carlo tree.

## 6 Conclusion

We have reviewed and evaluated S-PRNGs, and we have found statistical bias in most of them. As a main result, we have identified linear dependencies in the Monte Carlo trees generated by linear congruential generators, including both the Lehmer trees and the PRNG implemented in Haskell's standard library.

There is one S-PRNG construction where we have not been able to find any fault, namely the hash based generator of Claessen and Pałka (2013a). In particular, the generator was subject to the test methodology described in Sections 5.5–5.6, passing all tests. The foundation on cryptographic theory justifies some confidence in the construction. Even for this generator, the Birthday Paradox should be remembered and one should never use more than $O(\sqrt{n})$ states from a state space of size $n$.

S-PRNGs are essential for parallelising probabilistic algorithms in deterministic programs. An application of the hash-bashed S-PRNG to a data-parallel implementation of genetic algorithms was discussed by Schaathun (2014), a work which was motivated by a dynamic resource optimisation problem described by Bye and Schaathun (2014). It has been quite surprising to see how hard it is to find examples of parallel, probabilistic algorithms in pure functional programming, and as we have pointed out, the first convincing, publicly available implementation of a suitable S-PRNG came in 2013.

While the hash-based S-PRNG is very convincing, both theoretically and practically, one should hope to see more thorough, independent evaluations thereof. Conventional wisdom in computational statistics and Monte Carlo simulation indicate that one PRNG cannot suit every purpose. Different applications have different priorities, and calling for different trade-offs between sequence length, distribution, running time, and theoretical guarantees. Therefore, this topic is not exhausted yet.

The S-PRNG construction of Steele *et al.* (2014) was published after the original submission of this paper and has therefore been omitted from the study.

## Acknowledgement

The author is also very grateful to the three anonymous referees whose thorough reviews, encouragement, and very constructive criticism have been a major help in developing this paper.

## References

Brown, R. G. (2015 January) *Dieharder: A Random Number Test Suite*. Software. Available from `http://www.phy.duke.edu/~rgb/General/dieharder.php`.

Burton, F. W. & Page, R. L. (1992) Distributed random number generation. *J. Funct. Program.* **2**(2), 203–212.

Bye, R. T. & Schaathun, H. G. (2014) An improved receding horizon genetic algorithm for the tug fleet optimisation problem. In Proceedings of the 28th European Conference on Modelling and Simulation (ECMS 2014). ECMS European Council for Modelling and Simulation, pp. 682–690.

Carta, D. G. (1990) Two fast implementations of the minimal standard random number generator. *Commun. ACM* **33**(1), 87–88.

Claessen, K. & Pałka, M. H. (2013a) Splittable pseudorandom number generators using cryptographic hashing. In Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. Haskell '13. New York, NY, USA: ACM, pp. 47–58.

Claessen, K. & Pałka, M. H. (2013b) *The tf-Random Package*. Accessed 2015-01-15. Online at: `https://hackage.haskell.org/package/tf-random`.

Cuccaro, S. A., Mascagni, M., & Pryor, D. V. (1995) Techniques for testing the quality of parallel pseudorandom number generators. In Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing, Bailey, D. H., Bjørstad, P. E., Gilbert, J. R., Mascagni, M., Schreiber, R. S., Simon, H. D., Torczon, V. J., & Watson, L.T. (eds), SIAM, pp. 279–284.

De Matteis, A. & Pagnutti, S. (1990) Long-range correlations in linear and non-linear random number generators. *Parallel Comput.* **14**(2), 207–210.

Eddy, W. F. (1990) Random number generators for parallel processors. *J. Comput. Appl. Math.*, **31**(1), 63–71.

Frederickson, P., Hiromoto, R., Jordan, T. L., Smith, B. & Warnock, T. (1984) Pseudo-random trees in Monte Carlo. *Parallel Comput.* **1**(2), 175–180.

Hackage. (2011) *The Random Package*. Haskell Random Number Library. Documentation. Accessed 2015-01-16. Online at: `http://hackage.haskell.org/package/random-1.1`.

Halton, J. H. (1989) Pseudo-random trees: Multiple independent sequence generators for parallel and branching computations. *J. Comput. Phys.* **84**(1), 1–56.

Klamkin, M. S. & Newman, D. J. (1967) Extensions of the birthday surprise. *J. Comb. Theory* **3**(3), 279–282.

Knuth, D. E. (1998) *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley.

Koniges, A. E. & Leith, C. E. (1989) Parallel processing of random number generation for Monte Carlo turbulence simulation. *J. Comput. Phys.* **81**(1), 230–235.

Krawczyk, H. (1992) How to predict congruential generators. *J. Algorithms* **13**(4), 527–545.

L'Ecuyer, P. (1988) Efficient and portable combined random number generators. *Commun. ACM* **31**(6), 742–749 and 774.

L'Ecuyer, P. (2012) Random number generation. In *Handbook of Computational Statistics: Concepts and Methods*, Gentle, J. E., Härdle, W. K. & Mori, Y. (eds), 2nd ed., Springer, pp. 35–71.

L'Ecuyer, P. & Simard, R. (2007) TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.* **33**(4), Article no. 22.

L'Ecuyer, P., Simard, R., Chen, E. J. & Kelton, W. D. (2002) An objected-oriented random-number package with many long streams and substreams. *Oper. Res.* **50**(6), 1073–1075.

Leiserson, C. E., Schardl, T. B. & Sukha, J. (2012) Deterministic parallel random-number generation for dynamic-multithreading platforms. *ACM SIGPLAN Not.* **47**(8), 193–204.

Marlow, S. (2013) *Parallel and Concurrent Programming in Haskell.* O'Reilly.

Marsaglia, G. (1968) Random numbers fall mainly in the planes. *Proc.Natl. Acad. Sci. United States Am.* **61**(1), 25–28.

Mascagni, M. (1998) Parallel linear congruential generators with prime moduli. *Parallel Comput.* **24**(5–6), 923–936.

Matsumoto, M. & Nishimura, T. (1998) Dynamic creation of pseudorandom number generators. In Monte Carlo and quasi-Monte Carlo Methods, 1998: Proceedings of a Conference Held at the Claremont Graduate University, June 22–26, Niederreiter, H. & Spanier, J. (eds), Claremont, CA, USA: Springer, pp. 56–69.

Matsumoto, M., Wada, I., Kuramoto, A. & Ashihara, H. (2007) Common defects in initialization of pseudorandom number generators. *ACM Trans. Model. Comput. Simul.* **17**(4), Article no. 15.

Menezes, A. J., van Oorschot, P. C. & Vanstone, S. A. (1997) *Handbook of Applied Cryptography.* CRC Press.

O'Sullivan, B., Goerzen, J. & Stewart, D. (2008) *Real World Haskell.* O'Reilly.

Park, S. K. & Miller, K. W. (1988) Random number generators: Good ones are hard to find. *Commun. ACM* **31**(10), 1192–1201.

Percus, Ora E. & Kalos, M. H. (1989) Random number generators for MIMD parallel processors. *J. Parallel Distrib. Comput.* **6**(3), 477–497.

Salmon, J. K., Moraes, M. A., Dror, R. O. & Shaw, D. E. (2011) Parallel random numbers: As easy as 1, 2, 3. In High performance computing, networking, storage and analysis (SC11), 2011 International conference for. ACM, pp. 1–12.

Schaathun, H. G. (2014) Parallell slump (Om å parallellisera genetiske algoritmar i Haskell) *Norsk informatikkonferanse.* Open access at: `http://ojs.bibsys.no/index.php/NIK/index`. ISSN 1892–0721.

Steele, Jr., Guy L., Lea, D. & Flood, C. H. (2014) Fast splittable pseudorandom number generators. *ACM SIGPLAN Not.* **49**(10), 453–472.

Warnock, T. T. (1983) Synchronization of random number generators. *Congressus Numerantium* **37**, 135–144.

Wu, P.-C. & Huang, K.-C. (2006) Parallel use of multiplicative congruential random number generators. *Comput. Phys. Commun.* **175**(1), 25–29.