Splittable Pseudorandom Number Generators using Cryptographic Hashing

(article starts on next page)

Splittable Pseudorandom Number Generators using Cryptographic Hashing

Koen Claessen   Michał H. Pałka
Chalmers University of Technology
koen@chalmers.se   michal.palka@chalmers.se

Abstract

We propose a new splittable pseudorandom number generator (PRNG) based on a cryptographic hash function. Splittable PRNGs, in contrast to linear PRNGs, allow the creation of two (seemingly) independent generators from a given random number generator. Splittable PRNGs are very useful for structuring purely functional programs, as they avoid the need for threading around state. We show that the currently known and used splittable PRNGs are either not efficient enough, have inherent flaws, or lack formal arguments about their randomness. In contrast, our proposed generator can be implemented efficiently, and comes with a formal statements and proofs that quantify how 'random' the results are that are generated. The provided proofs give strong randomness guarantees under assumptions commonly made in cryptography.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Algorithms, Languages

Keywords splittable pseudorandom number generators, provable security, Haskell

1. Introduction

Splittable pseudorandom number generators (PRNGs) are very useful for structuring purely functional programs that deal with randomness. They allow different parts of the program to independently (without interaction) generate random values, thus avoiding the threading of a random seed through the whole program [10]. Moreover, splittable PRNGs are essential when generating random infinite values, such as random infinite lists in a lazy language, or random functions. In addition, deterministic distribution of parallel random number streams, which is of interest to the High-Performance Computing community [22, 26], can be realised using splitting.

In Haskell, the standard module `System.Random` provides a default implementation of a splittable generator `StdGen`, with the following API:

```
split :: StdGen -> (StdGen, StdGen)
next  :: StdGen -> (Int, StdGen)
```

The function `split` creates two new, independent generators from a given generator. The function `next` can be used to create one random value. A user of this API is not supposed to use both `next` and `split` on the same argument; doing so voids all warranties about promised randomness.

The property-based testing framework QUICKCHECK [13] makes heavy use of splitting. Let us see it in action. Consider the following simple (but somewhat contrived) property:

```
newtype Int14 = Int14 Int
  deriving Show

instance Arbitrary Int14 where
  arbitrary = Int14 `fmap` choose (0, 13)

prop_shouldFail (_, Int14 a) (Int14 b) = a /= b
```

We define a new type `Int14` for representing integers from 0 to 13. Next, we create a random generator for it that randomly picks a number from 0 to 13. Finally, we define a property, which states that two randomly picked `Int14` numbers, one of which is a component of a randomly picked pair, are always unequal.

Testing the property yields the following result:

```
*Main> quickCheckWith
  stdArgs { maxSuccess = 10000 } prop_shouldFail
+++ OK, passed 10000 tests.
```

Even though the property is false (we would expect one of every 14 tests to fail), all 10000 tests succeed!

The reason for this surprising behaviour is a previously unknown flaw in the standard Haskell pseudorandom number generator used by QUICKCHECK during testing. The PRNG should pick all combinations of numbers 0–13 for `a` and `b`, but in fact combinations where `a` and `b` are the same number are never picked.

It turns out that the `StdGen` standard generator used in current Haskell compilers contains an *ad hoc* implementation of splitting. The current implementation is the source of the randomness flaw[1] demonstrated above. The flaw requires a particular pattern of split operations to manifest and results in very strong correlation of generated numbers. In fact, when 13 in the `Int14` generator is replaced by other numbers from range 1–500, the problem arises for 465 of them! Unfortunately, this pattern of splits is simple and likely to arise often in typical usage of QuickCheck. Because of this, we cannot be sure that QuickCheck properties that pass a large number of tests are true with high probability.

Unfortunately, research devoted to pseudorandom generation has mainly concentrated on linear generators, which do not support on-demand splitting. Several attempts have been made at extending

[1] `http://hackage.haskell.org/trac/ghc/ticket/3575` and `.../3620`

linear PRNGs into splittable ones [10, 19, 29]. Most proposed constructions were incompatible with unlimited on-demand splitting. Yet the ones that supported it did not assure the independence of derived generators. In fact, the current implementation of splitting in `System.Random` contains a comment *'no statistical foundation for this!'*. Indeed.

Attempting to generalise an existing traditional linear PRNG into a splittable one is problematic for two reasons. First, it is not clear how randomness properties of a linear generator carry over to a splittable generator. For example, demanding that every path through a splitting tree is a good linear PRNG is not at all enough to ensure that the two subgenerators created by a split are independent. And second, which may even be more problematic, the formal requirements satisfied by traditional linear PRNGs are not even sufficient to guarantee good randomness. Instead, their randomness is assessed using suites of statistical tests [25, 34, 36]. However, even if a linear generator passes these tests, this may not guarantee that it will work well in particular situations, as some linear PRNGs have been found to fail in intricate ways [30]. So, because of the lack of strong formal properties satisfied by linear PRNGs, there is no reliable way of extending a linear PRNG into a splittable one that guarantees good statistical properties.

In contrast, cryptographic research has resulted in methods for generating high-quality pseudorandom numbers, which are provably random, using pseudorandom functions (PRFs) [5, 20, 24, 31]. The proofs depend on unproven but commonly accepted assumptions, such as the computational difficulty of some problems, or on the existence of secure block ciphers. Despite that, they provide a much higher degree of confidence than statistical test suites. Even more importantly, the proofs serve as guidance for deciding which constructions of PRNGs are sound.

Cryptographic methods have been proposed for implementing splittable PRNGs recently. In a Haskell-Cafe mailing list discussion [32] Burton Smith et al. propose basing such a generator on a cryptographic block cipher. Another example are random number generators specified in NIST SP 800-90A [2], which are based on PRNGs (called DRBGs[2] there), whose one instance can be initialised using pseudorandom output from another instance. This mechanism has been used for implementing splitting in the crypto-api Hackage package[3].

The idea behind both of these designs is similar and appears to give good results. However, only an informal justification is provided for the first of them, and none for the second. Furthermore, splitting is costly in both of them, as every `split` operation requires one (or more) run of the underlying cryptographic primitive, such as a block cipher.

An efficient PRNG that supports splitting under the hood has been proposed by Leiserson et al. [26]. To generate random numbers, the generator hashes a path, which identifies the current program location, using a hash function. The hash function is constructed in a way that minimises the probability of collisions and contains a final mixing step to make the output unpredictable. However, low probability of collisions is the only property that is proven for the hash function, whereas randomness of its results is only evaluated using statistical test.

Micali and Schnorr [31] present a PRNG based on the RSA cryptosystem, which is provably random and supports n-way splitting. However, the results are asymptotic, which means that they do not indicate what parameters to choose to achieve a particular level of randomness, or whether the generator is practical at all [12, 18, 35].

[2] Deterministic Random Bit Generators

[3] `http://hackage.haskell.org/packages/archive/crypto-api/0.12.2.1/doc/html/Crypto-Random.html`

Thus, until now all splittable PRNGs were either *ad hoc* extensions of linear PRNGs, informally justified solutions based on cryptographic primitives, or cryptographic constructions covered by asymptotic arguments, which do not say anything about their practical quality. In this paper, we propose an efficient, provably-random splittable PRNG and give concrete security bounds for it. Our generator is based on an existing cryptographic keyed hash function [5], which uses a block cipher as the cryptographic primitive. The generator is provably-random, which makes use of previously established concrete security proof of the hash function. Our contributions are as follows:

- We propose a splittable PRNG that provides provable randomness guarantees under the assumption that the underlying block cipher is secure. The construction is conceptually very simple, and relies on a known keyed hash function design. (Section 3)
- We provide proofs of randomness of the proposed generator, which rely on previously known results about the hash function. The randomness results are concrete, not asymptotic, and degrade gracefully if the assumptions about the block cipher are relaxed. (Section 4)
- We present benchmark results indicating that QUICKCHECK executes typical properties about 8% slower than with `StdGen`, and the performance of linear random number generation is good. (Section 6)
- We show that the problem solved by a splittable PRNG is essentially the same as the one solved by a keyed hash function. (Section 7)
- The obtained randomness bounds are weaker than those possible for a linear PRNG. So, we quantify the price in randomness we have to pay for the increased flexibility that splitting provides. (Section 7)

2. Splittable PRNGs

The traditional way of using random numbers is to give a program access to a process that generates a linear sequence of random numbers. This, however, is not a modular approach. Consider that we have function `f` that calls two other functions `g` and `h` that perform subcomputations.

```
f x = ... (g y) ... (h z)
```

Normally, when using a linear pseudorandom generator, `f` would pass a reference to the generator to `g`, which would consume some number of random numbers and change the state of the generator, and then to `h`, which would do the same.

There are two problems with this approach. First, it would create an unnecessary data dependency between `g` and `h` which may otherwise have been independent computations, destroying opportunities for possible laziness or parallelism. Secondly, any change to `g` that would influence the amount of random numbers that it consumes would also influence the computation of `h`. Thus, repeating the computation with the same random seed and a changed program would also introduce disturbances in unchanged places.

Addressing these problems is the goal of splittable PRNGs [10]. Consider the following interface for a PRNG whose state is represented by the type `Rand`. This API is simpler than the one presented in the introduction in that the linear `next` operation is replaced by `rand` that only returns one random number. We will come back to the original API in Section 5.

```
split :: Rand -> (Rand, Rand)
rand  :: Rand -> Word32
```

Operation `split` takes a generator state and returns two independent generator states, which are derived from it. Operation `rand` generates a single random number from a generator state. Both `split` and `rand` are *pure*, which means that given the same arguments they yield the same result. Calling `rand` many times with the same generator state will yield the same result each time. The intended way of using such a generator is to start with an initial value of `Rand` and then split it to generate many random numbers.

Similar to the original API, an additional requirement is placed on the program that it is not allowed to call both `split` and `rand` on a given generator state, since the parent generator state is not guaranteed to be independent from generator states derived from it.

When using this API, function `f` can simply call `split` and pass two independent derived generator states to its subcomputations, which may use them without any data dependencies. This allows for generating random lazy infinite data structures in an efficient way, as independent subcomputations can be launched at any time.

2.1 Naïve approach

A naïve way of implementing a referentially-transparent splittable PRNG is to start with a linear PRNG and make `split` divide the sequence of random numbers in some way [10]. For example, `split` may return the even-indexed elements as the left sequence and odd-indexed elements as the right sequence, whereas `rand` may return the first number of the sequence.

$$\texttt{split}\ (\langle x_0, x_1, \ldots\rangle) = (\langle x_0, x_2, \ldots\rangle, \langle x_1, x_3, \ldots\rangle)$$

Similarly, `split` may divide the whole sequence period into two halves and return them as subsequences. However, both of these approaches allow for a very short sequence of `split` operations, as after n splits the resulting sequence would be of length one, assuming the state size of n bits. Moreover, the amount of memory required to store a generator grows at least linearly in the number of splits.

Furthermore, there are no guarantees that the resulting numbers have good randomness properties. Traditionally, linear PRNGs are evaluated in terms of their periods, which is how long a sequence they can generate without experiencing a state collision. Making statements about periods is meaningless for a splittable generator; for any generator that uses a constant amount of n bits of memory, there will be two identical generator states at most n splits away from each other, given that a full binary tree of depth n has 2^n leaves. The precise overall randomness properties have not been formalised for traditional linear PRNGs, and thus cannot be carried over to splittable generators at all.

An improved idea for a splittable PRNG is that the `split` operation jumps to a *random* place in the linear sequence [10]. The design is compelling, as it does not suffer from the obvious limitation that concerns regular ways splitting. However, the quality of returned random numbers would again depend on the suitability of the original sequence, which is hard to determine.

Another approach for distributing unique generator states to subcomputations is to use an impure operation under the hood that returns numbers from a pseudorandom sequence, similarly to this solution for generating unique names [1]. However, such generator would no longer give deterministic results if the order of evaluation changes.

2.2 Pseudorandomness

We found that a formal definition of pseudorandomness is essential for designing and evaluating a splittable PRNG. In particular, we found that the concept of pseudorandomness used in cryptography matches closely our goal of creating random-looking numbers, which we explain in this section.

For simplicity, we first consider linear pseudorandom number generation. Consider non-deterministic program D that can perform a number m of coin flips and use their results in computations. We can model such a program with a deterministic function that has access to a sequence of m bits that has been chosen uniformly at random (we will use bits instead of numbers for simplicity). We assume that all 'deterministic' inputs are already baked into the program, and that it returns '0' or '1'.

We quantify the behaviour of the non-deterministic program D by the probability that it returns '1' when run with a sequence of bits chosen uniformly at random.

$$\Pr_{r \leftarrow \{0,1\}^m}[D(r) = 1]$$

The goal of pseudorandom number generation is to replace the randomly chosen sequence of bits by one generated by deterministic function p (the generator) from a small seed ($s \in S = \{0,1\}^n$) chosen at random, so that the program gives almost the same results with both random sequences. Thus, the probabilities of returning '1' by program D should be close to each other when run with a fully random sequence and with a pseudorandom sequence generated by function p. The absolute difference in these probabilities is D's *advantage* in distinguishing the output of p from random bits.

$$\mathrm{Adv}_D(p) = \left| \Pr_{r \leftarrow \{0,1\}^m}[D(r) = 1] - \Pr_{s \leftarrow S}[D(p(s)) = 1] \right|$$

Unfortunately, when $m > n$, it is always possible to construct a 'distinguishing' program that will make the difference in results large, since the image $p[S]$ is only a small part of all possible sequences. If program D checks whether the sequence of bits belongs to $p[S]$ outputting '1' in that case and '0' otherwise, then its results would always be '1' with p, but only half of the time or less with fully random bits.

Thus, any pseudorandom number generator p can be distinguished from a fully random choice of bits in the sense of information theory, that is when the distinguishing program has unlimited resources. However, it is widely believed that it is possible to construct effectively computable p, whose output can be distinguished from a true random sequence by a non-negligible margin only by programs that perform an enormous amount of computation [8, 20, 24, 36].

It has been shown that such deterministic functions whose output appears to be random (in an asymptotic sense) to any polynomial-time program can be constructed from one-way functions [20, 24]. At the same time, the existence of one-way functions is considered to be a reasonable complexity-theoretic assumption. For example, some number-theoretic functions, such as the RSA-function, are believed to be one-way [4].

However, PRNG constructions based on one-way functions are inefficient. Cryptographic *block ciphers*, on the other hand, offer a more efficient alternative for implementing PRNGs [5, 34]. Pseudorandomness of such generators depends on the security of block ciphers they are based on.

Block ciphers are common and widely used cryptographic primitives. Unlike number-theoretic constructions, the security of block ciphers does not follow from complexity-theoretic assumptions. Instead, their security is *asserted* based on careful design and unsuccessful cryptanalytic attempts on them [4, 33]. Even though there are many block ciphers that had once been considered safe and were subsequently broken, there exists a selection of them that successfully underwent a thorough peer-review process without being broken, and are generally considered to be trusted primitives.

Asserting the security of a block cipher appears to be an unnecessarily strong assumption, given that constructions based on one-way functions could be used. However, proving their concrete (non-asymptotic) pseudorandomness requires making additional concrete assumptions about the one-way functions, which are much less obvious.

We propose to use the 256-bit variant of the ThreeFish [17] block cipher for the construction of our PRNG, which is one such peer-

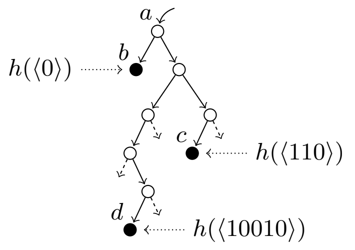


Figure 1. Splittable PRNG through hashing of paths.

reviewed primitive. It has been proposed as the basis for the Skein hash function, which was one of the finalists of the National Institute of Standards and Technology (NIST) SHA-3 secure hash standard competition[4]. The hash function survived three rounds of public peer-review, withstanding the best attacks with a reasonable security margin, and its security was considered acceptable by NIST [11].

3. Proposed construction

We propose a simple construction that gives a splittable PRNG, whose output appears to be fully random to any program that performs a reasonably bounded amount of computation. A program that has access to such a generator, using the API presented in Section 2, can use splitting to create a number of generator states over the course of its execution, and can also query some of these states for random numbers using `rand`. Observe that we can identify each generator state with the path that was used to derive it from the initial generator state. The main idea behind our construction is to map these paths to numbers using a keyed hash function, whose results for different arguments appear to be unpredictably random.

First, we injectively encode each path as a sequence of bits. Starting from the initial generator state we take '0' each time we go to the left and '1' to the right. Then we use a cryptographic keyed hash function to map that sequence of bits to a hash value. The initial seed of the generator is used as the key for the hash function. The result of the hash function is our random number.

Figure 1 shows a tree of generator states created by splitting the initial generator state a. Random numbers are generated by computing the hash function of the encoded paths leading to the respective generator states. For example, generator state c has been obtained from the initial state using 3 `split` operations: first the right generator state has been chosen, second also the right one, and third the left one. Hence, the path leading to c is encoded as bit sequence $\langle 110 \rangle$, and the hash of that is the random number returned by `rand` c.

The following code shows a simple, but inefficient Haskell implementation of this generator.

```
type Rand = (Seed, [Bit])

split :: Rand ->  (Rand, Rand)
split (s, p) = ((s, p ++ [0]), (s, p ++ [1]))

rand :: Rand -> Word32
rand  (s, p) = extract $ hash s p
```

The concern of this simple construction is to uniquely encode each derived generator state. The entire task of creating random-looking output is outsourced to the keyed hash function, which we chose to implement with a tried and tested construction.

[4] `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`

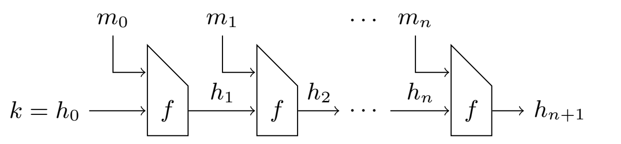


Figure 2. Merkle-Damgård construction. f represents the compression function.

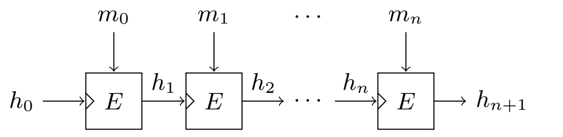


Figure 3. Merkle-Damgård construction using a block cipher. Input indicated with a triangle is the key of the block cipher.

3.1 Efficient hashing

Using a cryptographic hash function to process the paths leading to every random number may seem to be very slow. However, cryptographic hash functions often have an iterated construction, which allows them to be computed incrementally. By doing so we can implement both `split` and `rand` in $O(1)$ time, and make `split` a very cheap operation most of the time. In this way we achieved performance that almost matches the current standard splittable generator for Haskell.

We use the Merkle-Damgård construction [5, 15, 16] to implement hashing, which is a common pattern for iterated hash functions. In this construction, shown in Figure 2, the input consists of a number of fixed-width data blocks m_0 to m_n that are iteratively fed into function f (the *compression* function), which takes both a data block and intermediate state h_i and computes new intermediate state h_{i+1}.

$$h_{i+1} = f(h_i, m_i)$$

The hash key is used as the initial state of the iteration h_0, and the final state h_{i+1} is the value of the hash. The function is essentially a *fold* of the compression function f, and can be computed incrementally.

$$\texttt{hash}\ k\ [m_0, \ldots, m_n]\ \texttt{=}\ \texttt{foldl'}\ f\ k\ [m_0, \ldots, m_n]$$

The purpose of the compression function is to mix a block of input data into an intermediate state in an unpredictable way.

The unpredictability can only be achieved when the initial state and all intermediate states are unknown to the program receiving the outputs of the hash function. For this reason, none of the inputs for the hash function with a given key should be a prefix of another one. Otherwise, the result of the hash function for the shorter input would be the same as one of the intermediate states for the longer one, which would mean that one function result could be predicted from another. In other words, the construction requires that the set of queried inputs is *prefix-free*.

The construction relies on the compression function for unpredictability, merely extending its domain from single blocks to their sequences. The compression function can be implemented using a block cipher by feeding the state h_i as the key of the cipher and the data blocks as the data blocks of the cipher, as shown in Figure 3.

The requirements of the hash function that the input is a sequence of blocks of a size determined by the block cipher, and that the inputs used during one program run form a prefix-free set, need to be addressed by the encoding used to create inputs. Below we present such an encoding.

3.2 Encoding

We encode the paths leading to generator states as sequences of blocks. First, let's represent a path as a sequence of bits as previously, by walking the path from the initial generator state and taking '0' for going left and '1' for going right. Then, we divide this sequence into blocks of the required size and if the last block is incomplete, pad it with '0's.

Observe that the paths to the generator states that are queried in a single run of a program must form a prefix-free set, due to the requirement that `split` and `rand` cannot be called on the same generator state.

The fact that the paths form a prefix-free set is required for the encoding to be injective. To show the injectivity, let's assume the opposite. Suppose that we have two different paths that are mapped to the same sequence of k blocks. Given that the initial $k-1$ blocks are equal, the paths must have a common prefix that was encoded in these blocks. Since the last block is also equal, the two paths must have different lengths and the suffix of the longer one must be encoded as all '0's to match the padding in the encoding of the shorter one. Furthermore, the part that precedes that suffix must be equal to the last part of the shorter path. From this we can see that the shorter path must be a prefix of the longer one, which contradicts our assumption.

We can similarly show that no sequence of blocks is a prefix of another one if both are encodings of paths that are not each other's prefixes. Thus, the encodings of paths that form a prefix-free set will themselves form a prefix-free set, which is required by the hash function.

The encoding is efficient in the sense that to hash a given path a block cipher is invoked only once every b splits where b is the block size in bits.

3.3 Incremental computation

To obtain hashing and encoding presented above in an efficient and incremental way we implemented both of these stages together in the `split` and `rand` operations. Operation `split` takes a partially-computed hash value and returns two partially-computed hash values. Operation `rand` completes the computation of the hash and returns its value. Implementing the encoding and hashing together has the added benefit that the hash computation of the initial common part of the path can be shared between different generator states.

To allow incremental computation the generator state must contain both the last intermediate state block of the hash function and the last part of the path that has not been hashed yet. Operation `split` adds a bit ('0' in the right generator state, '1' in the left one) to the unhashed sequence and if the unhashed sequence has reached the length of a block, it runs one iteration of the hash function that consumes the sequence and updates the intermediate state. Operation `rand` runs the final iteration of the hash function with whatever unhashed sequence there is (it might be empty), pads it with '0's and extracts a random number from the hash.

```
type Rand = (Block, [Bit])

hashStep :: Rand -> Rand
hashStep s@(h_i, unhashed)
  | length unhashed < blockLength = s
  | otherwise = (compress h_i (pad unhashed), [])

split :: Rand -> (Rand, Rand)
split (h_i, unhashed) =
  (hashStep (h_i, unhashed ++ [0]),
   hashStep (h_i, unhashed ++ [1]))

rand :: Rand -> Word32
rand (h_i, unhashed) =
  extract $ compress h_i (pad unhashed)
```

Since the length of `unhashed` is at most equal to `blockLength`, the run time of both `rand` and `split` operations is bounded. Also, `split` is usually inexpensive, as most of the time it only adds a single bit to the state.

4. Correct hashing

The Merkle-Damgård hashing construction shown by us in the previous section relies on a block cipher, or another compression function with similar properties, to provide pseudorandomness. However, the block cipher invocations are chained, making its use rather non-trivial. Thus, it is important to consider the correctness of the construction.

One way of analysing a cryptographic hash function is to use a security reduction proof to bound the advantage of any program in distinguishing the results of the hash function from completely random results, depending on the amount of computation the program can perform. Proving the hash function's correctness amounts to showing that it gives only negligible advantage to programs that perform reasonable amount of computation.

The M-D construction has previously been analysed from this perspective. Below, we present a reduction proof of its pseudorandomness, based on [4, 5]. the proof allows us to derive concrete bounds on the maximum discrepancy that a program using the generator can observe.

Using security proofs turned out to be very helpful in designing the PRNG, as they helped to identify incorrect designs as well as their elements, which were not bringing any benefits. In addition, the bounds that they provide made the limitations of the generator explicit.

The reduction proof assumes that the used block cipher is secure in the sense that it can be effectively broken only by key enumeration, which is the main assumption of our PRNG (see Section 2.2). Moreover, even if the cipher's known security is reduced, the bound provided by the proof degrades gracefully.

However, perhaps the biggest value of the proof is the guarantee that the block cipher is used in a correct way in the construction, as otherwise it would not have been possible to achieve a low bound on the discrepancy.

4.1 Pseudorandom functions

In order to formally model block ciphers and keyed hash functions, we need to introduce pseudorandom functions (PRFs) [4, 5, 20], which generalise pseudorandom number generators.

Let $F = \{f_k : k \in K\}$ be a finite family of functions $f_k \in B^A$ indexed by elements of finite set K. We write $f \leftarrow F$ to mean that f has been chosen from F by choosing its index uniformly at random from K. Next, consider $F' = B^A$ to be the set of functions from A to B, where B is a finite set, and A is possibly infinite.

This introduces a slight complication, as F' may be infinite, and the uniform distribution does not exist for coutably infinite sets. To address that, we use the *lazy sampling* principle [3]. That is, instead of choosing a random member of B^A, we create a lazy non-deterministic procedure that generates the function's random results on demand. The procedure picks a random value of B each time the function is called with a new argument. Consequently, the results of the function appear exactly as if they were returned by a 'randomly chosen' function. We use the same notation $f' \leftarrow F'$ to mean that f' has been chosen from F' using lazy sampling. For function families that are finite, when both A and B are finite, lazy sampling gives the same observed distribution as choosing their representant uniformly at random.

We generalise the definition of advantage from Section 2.2. Let F_1 and F_2 be two function families of functions B^A, each of which may be an indexed function family, or the set of all functions from

A to B. The *advantage* of D in distinguishing between F_1 and F_2 is defined as follows:

$$\mathrm{Adv}_D(F_1, F_2) = \left| \Pr_{f_1 \leftarrow F_1}[D(f_1) = 1] - \Pr_{f_2 \leftarrow F_2}[D(f_2) = 1] \right|$$

where $D(f)$ is program D instantiated with a particular function $f : B^A$, which it can query as a black-box oracle.

Our objective now will be to have a small and easily computable indexed function family $F \subseteq B^A$, whose random member appears to programs as a random function B^A. In other words, the advantage $\mathrm{Adv}_D(F, B^A)$ should be as small as possible for any (reasonably bounded) D, making F and B^A difficult to distinguish (*indistinguishable*). Such an indexed function family is said to be *pseudorandom*.

The original goal of bounding the advantage of any program with a certain run time limit has been to defend against an adversary who could create a program that breaks the pseudorandom number generator, or another cryptographic entity. Even though we do not assume any adversarial behaviour, we would like that the user of a PRNG can write any program without risking that a flaw in the generator will compromise his results.

4.2 Iterative hash construction

We will now formalise the construction of the hash function presented in Section 3.1. The hash function uses a compression function f to process each of the data blocks in turn. Let $F = \{f_k : k \in B\}$ be an indexed function family representing the compression function, so that $f_k(x) = f(k, x)$ and $B = \{0, 1\}^n$ represent the set of data blocks, as well as the set of keys. We define $F^* = \{f^*_k : k \in B\}$ with $f^*_k \in B^{B^*}$ to be a finite function family that represents the iterative hash construction based on f, as follows by induction.

$$\begin{aligned} f^*_k(\langle\rangle) &= k \\ f^*_k(\langle B_0, \ldots, B_{n-1}, B_n\rangle) &= f_{f^*_k(\langle B_0, \ldots, B_{n-1}\rangle)}(B_n) \end{aligned}$$

To show that this iterative hash construction is a good pseudorandom function we show a bound on the advantage program D can get in distinguishing it from a random function.

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leq \epsilon$$

The above bound can only be shown assuming that is that the advantage in distinguishing the compression function from a random function ($\mathrm{Adv}_{D'}(F, B^B)$) is also bounded. The proof also captures the crucial assumption that the sequences used by D for making queries form a prefix-free set.

To show the bound we refer to a result from [5]. The following theorem assumes the Random-Access Machine (RAM) computational model, which we also assume for the rest of this section.

Theorem 4.1. *Let F be a finite function family $F = \{f_k : k \in B\}$ with $f_k : B \to B$. Suppose that there exists program D, which has access to an oracle of type $B^* \to B$, and makes at most q prefix-free queries to this oracle at most $l \geq 1$ blocks long, consuming at most t units of time*[5]*. Then, there exists program E, which has access to an oracle of type $B \to B$, as well as to program D as another oracle, such that:*

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leq ql\, \mathrm{Adv}_{E(D)}(F, B^B)$$

Program $E(D)$ makes at most q queries and runs in time at most $t + cq(l + k + b)(\mathrm{Time}(F) + \log q)$, where $\mathrm{Time}(F)$ is the time required to execute the implementation of F and c is a small constant.

Proof. The theorem follows from Theorem 3.1 in [5]. □

[5] Time to read the program text is also counted in t.

The theorem states that if there exists program D that distinguishes between F^* from a random function with probability ϵ, then it is possible to construct program $E(D)$, which uses D as its subprogram, and that is able to distinguish the compression function F from a random function with probability $ql\epsilon$, with additional constraints on the run time of both programs.

Conversely, if we assert that the advantage of any program in distinguishing F^* from a random function is not greater than ϵ, then we can bound the advantage of any program for distinguishing F from a random function by $ql\epsilon$.

It is worth noting that we only specified the computational model for the reduction as the RAM computational model, and did not precise what additional operations programs may execute, such as the decryption function of a block cipher. This may seem suspicious as using a computational model that is weaker may appear to 'leave out' some ways to attack the hash. However, in fact using a weaker model can only result in $\mathrm{Adv}_{E(D)}(F, K^B)$ being smaller and providing a more conservative bound on $\mathrm{Adv}_D(F^*, K^{B^*})$.

To use Theorem 4.1 to bound the advantage on the hash function, we need to supply the bound on the compression function.

4.3 Compression function

In order to obtain an unpredictable compression function, we employ a cryptographic *block cipher*. A block cipher is a pair of effectively computable functions $\mathsf{enc} : K \times B \to B$ and $\mathsf{enc}^{-1} : K \times B \to B$, where K is the set of keys and B is the set of blocks. Both keys and blocks are fixed-length sequences of bits, and for our purposes we assume that $K = B = \{0, 1\}^n$. Each $\mathsf{enc}_k : B \to B$ is a permutation on B and also $(\mathsf{enc}_k)^{-1} = \mathsf{enc}^{-1}_k$.

The theoretical model for block ciphers that we will use is pseudorandom permutations (PRPs) [6, 33]. A PRP is an indexed function family that is indistinguishable from a random permutation in the same way as a pseudorandom function is indistinguishable from a random function.

To show the security bound of the hash function we need the compression function to be a PRF, but a block cipher that we have is a PRP. However, using a PRP in place of a PRF does not impose a large penalty on observed randomness, which resulted in many analyses treating block ciphers as PRFs [3].

The 'similarity' of PRPs and PRFs is stated by the PRP/PRF Switching Lemma [3, 21], which gives a bound on the ability to differentiate between a PRF and a PRP by a computationally-unbounded program ($\mathrm{Perm}(B)$ denotes the set of permutations of B).

$$\mathrm{Adv}_D(B^B, \mathrm{Perm}(B)) \leq \frac{q(q-1)}{2^{b+1}}$$

where $B = \{0, 1\}^b$ and q is the number of queries D can make to the oracle. Thus, the best way of distinguishing a PRP from a PRF is to query it a large number of times and look for collisions.

Given that, we can obtain a bound on expression $\mathrm{Adv}^D_E(F, B^B)$ from Theorem 4.1. From the definition of Adv we have the inequality:

$$\begin{aligned} \mathrm{Adv}_{E^D}(F, B^B) \leq \mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B)) + \\ \mathrm{Adv}_{E^D}(\mathrm{Perm}(B), B^B) \end{aligned}$$

Now we can bound $\mathrm{Adv}_{E^D}(\mathrm{Perm}(B), B^B)$ using the Switching Lemma:

$$\mathrm{Adv}_{E^D}(B^B, F) \leq \mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B)) + \frac{q(q-1)}{2^{b+1}}$$

To bound $\mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B))$, we have to invoke our assertion that F is a good block cipher. It is reasonable to assume that the best attacks on F are by key search [4].

$$\mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B)) \leq 2^{-b}\left(q + \frac{t'}{\mathrm{Time}(F)}\right)$$

Finally

$$\mathrm{Adv}_{E^D}(F, B^B) \leq 2^{-b}\left(q + \frac{t'}{\mathrm{Time}(F)} + \frac{q(q-1)}{2}\right)$$

Now invoking Theorem 4.1, we get:

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leq$$
$$2^{-b}ql\left(q + \frac{t + cq(l+k+b)(\mathrm{Time}(F) + \log q)}{\mathrm{Time}(F)} + \frac{q(q-1)}{2}\right)$$

After simplification and omitting insignificant terms we obtain the final bound:

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leq 2^{-b}\left(\frac{q^3l}{2} + c_1q^2l^2(1+\log q) + \frac{c_2tql}{\mathrm{Time}(F)}\right)$$

For example for a 256-bit block cipher, if program performs at most 2^{50} queries, each of at most 2^{30} blocks, then the discrepancy can be bounded by 2^{-70}, provided that the run time is bounded by 2^{90}. We assume that the constants are small ($c_2/\,\mathrm{Time}(F) < 2^{10}$).

5. Linear generation

We presented only a simplified version of splittable pseudorandom number generators with `split` and `rand` operations. Splittable generators in Haskell also support linear generation, which is possible with a modified API:

```
split :: Rand -> (Rand, Rand)
next  :: Rand -> (Word32, Rand)
```

Operation `next` replaces `rand` and, in addition to returning a random number, also returns a new generator state. Calling `next` repeatedly allows generating a sequence of random numbers. The API does not allow calling both `split` and `next` on the same generator state, similarly as in the original API presented earlier.

Replacing `rand` with `next` does not give any additional expressiveness to the API, as `next` can be implemented on top of the old API in the following way:

```
next g = (rand g1, g2)
  where
  (g1, g2) = split g
```

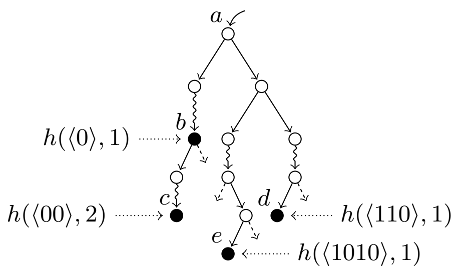
However, having `next` allows implementing linear random number generation more efficiently by counting the number of `next` operations on the path (which takes only $O(\log n)$ bits), in addition to keeping the sequence of `right` and `left` operations. Figure 4 presents encodings of paths from the initial state a to several derived generator states (marked in black). For example, path to state d is encoded as $(\langle 110\rangle, 1)$, because it contains one occurrence of `next` and the remaining operations on the path form the sequence `right`, `right`, `left`. Observe that it is possible for another path to be encoded as $(\langle 110\rangle, 1)$. However, reaching both that other state and d would require calling both `split` and `next` on some state, and thus violating the API requirement.

The used encoding (`e1`) is defined below. We formally show its uniqueness, together with a stronger property, which is of interest for creating more advanced encodings.

Definition 5.1. Encoding `e1` is a function that maps a path of three operations: `right`, `left` and `next` to a pair of a sequence of bits and a number. The number is equal to the number of `next` operations and the sequence of bits represents the sequence of `right` and `left`.

The following definition captures the requirement that must be satisfied by programs using the splittable PRNG API.

Definition 5.2. A program run that uses the API is *valid* iff, for any two distinct generator states queried in a single program run, the following holds. Let paths c_1 and c_2 represent the two states. Let



a
b
c
d
e
$h(\langle 0\rangle, 1)$
$h(\langle 00\rangle, 2)$
$h(\langle 110\rangle, 1)$
$h(\langle 1010\rangle, 1)$

Figure 4. Splittable PRNG with `next` operation. Snake arrows ($\rightsquigarrow$) lead to states derived using `next` operation.

c_p be their longest common prefix, and c_1' and c_2' be the respective remainders of c_1 and c_2. The following condition must be satisfied

$$\begin{aligned} & c_1' = \langle\rangle && \wedge\, c_2' = \langle \mathtt{next}, \ldots\rangle \\ \vee\, & c_1' = \langle \mathtt{next}, \ldots\rangle && \wedge\, c_2' = \langle\rangle \\ \vee\, & c_1' = \langle \mathtt{right}, \ldots\rangle && \wedge\, c_2' = \langle \mathtt{left}, \ldots\rangle \\ \vee\, & c_1' = \langle \mathtt{left}, \ldots\rangle && \wedge\, c_2' = \langle \mathtt{right}, \ldots\rangle \end{aligned} \tag{5.1}$$

where $\langle\rangle$ denotes the empty path, and '...' any remainder of a path.

To be able to state the main property of encoding `e1` we define relation $\precsim\!\succsim$. Let p_1 and p_2 be sequences; then $p_1 \precsim\!\succsim p_2$ means that one of p_1 and p_2 is a prefix of the other one. That is,

$$p_1 \precsim\!\succsim p_2 \quad \text{iff} \quad p_1 \preceq p_2 \vee p_2 \preceq p_1.$$

Note that $\preceq$ is reflexive, hence $\precsim\!\succsim$ is also reflexive. We will use $\precsim\!\succsim$ as a relation on both, sequences of bits and sequences of blocks.

The uniqueness of encoding `e1` for all states reachable from a single program run is implied by the following, stronger property.

Proposition 5.3. *If a program run is valid by Definition 5.2, then, for any two distinct generator states queried in the run, the following holds. Let paths c_1 and c_2 represent the two states, and (p_1, n_1) and (p_2, n_2) be their encodings using encoding `e1`. Then*

$$p_1 \not\precsim\!\succsim p_2 \vee n_1 \neq n_2 \tag{5.2}$$

Proof. Consider a program run that is valid according to Definition 5.2. Then, if c_1 and c_2 are paths to two queried states, then c_1' and c_2' defined as in Definition 5.2 satisfy Condition 5.1 stated there. We perform case analysis on the alternatives of that condition.

If any of the two initial alternatives is satisfied, then there is at least one more occurrence of `next` in one of c_1 and c_2 compared to the other one, hence $n_1 \neq n_2$, which means that the second alternative of Condition 5.2 is satisfied.

If any of the two final alternatives is satisfied, then $p_1 \not\precsim\!\succsim p_2$. □

5.1 Concrete encoding

We use the following scheme (e2) to encode the bit sequence and the counter, obtained from encoding `e1`, into a sequence of blocks. Zero or more initial blocks are devoted to encoding the bit sequence, while the last block encodes the counter. The following diagram shows the data layout of a sequence of blocks that encodes a path.

bseq$_1$	0	···	bseq$_n$	0	ctr	1

The bit sequence runs through the bseq$_i$ regions from front to back and is padded with '0's in the last region if it does not occupy the whole of it. The region in the last block marked ctr contains a binary number representing the counter. The last bit of the last block is set to '1' and to '0' for all other blocks.

We ignore the situation where the value of the counter is too large to be represented by the ctr field, and leave the encoding undefined in that case. We deal with overflow in a more complex encoding that is presented in Appendix A.

Note that linear random number generation using `next` requires only one iteration of the hash function as only the value of the counter is changing, which requires changing only the last block. This way, when `next` is called repeatedly, the block cipher used as the compression function is effectively run in counter mode for generating random numbers. Splitting, on the other hand, requires rehashing two blocks at the end. The more efficient encoding that is presented in Appendix A usually requires updating only the last block on during splitting.

5.2 Compatibility with hash function

To be compatible with the hash function presented in Section 3.1, the encoding must transform any set of paths, which are reachable in a single program run that satisfies the restriction imposed by the API, into a prefix-free set of blocks. Below, we show that this property holds for paths for which the encoding is defined.

Recall that, from Proposition 5.3, if (p_1, n_1) and (p_2, n_2) are two different paths that are inspected in the same program run encoded with e1, then the following holds:

$$p_1 \not\approx p_2 \vee n_1 \neq n_2$$

We must show that if (p_1, n_1) and (p_2, n_2) satisfy the above condition, then their encodings with e2 are not each other's prefixes. We prove this fact by contrapositive. Without loss of generality, we assume that $s_1 \preceq s_2$, where $s_1 = \mathsf{e2}\ (p_1, n_1)$ and $s_2 = \mathsf{e2}\ (p_2, n_2)$. The last bit of each block of s_1 and s_2 is set to '0', except for the last block, where it is set to '1'. Therefore, the last block of s_1 cannot be equal to a non-terminal block of s_2, hence $s_1 = s_2$. The counter is uniquely determined by the ctr region, therefore $n_1 = n_2$. The sequence of bits encoded with e2 is also uniquely determined, except for a number of possible trailing '0', which are indistinguishable from padding. Thus, p_1 and p_2 differ only by the amount of trailing zeroes, hence $p_1 \approx p_2$, which gives us $\neg(p_1 \not\approx p_2 \vee n_1 \neq n_2)$.

5.3 n-way split

The design of the generator suggested one more primitive operation `splitn`:

```
splitn :: Rand -> Word32 -> Rand
```

Calling `splitn g` yields 2^{32} new generator states derived from g, which can be accessed by applying the resulting function to numbers $0 \ldots 2^{32} - 1$. Consistently with the original API, we require that only one of the operations `next`, `split` and `splitn` can be called on a given generator state.

The n-way split operation can be used to efficiently create many derived generator states when their number is known in advance. For example, an array of random numbers can be efficiently generated using `splitn`. Another scenario when `splitn` might be useful is generating random functions in QUICKCHECK, which uses a sequence of `split` operations to derive a generator state, which is uniquely determined by an argument that was applied to the random function.

Semantically, `splitn` can be expressed in terms of a sequence of 32 `split` operations, and selecting one of the 2^{32} possible derived states, determined by the `splitn` argument. An efficient implementation, however, would instead add all 32 bits at once to the bit sequence that encodes the path to the generator state.

6. Performance

Cryptographic techniques have long been known for producing high-quality random numbers. However, their perceived low performance has been a barrier for their adoption. The original goal for this work has been to assess whether a splittable PRNG based on a block cipher can give acceptable performance so that it can be proposed to be the default PRNG in Haskell.

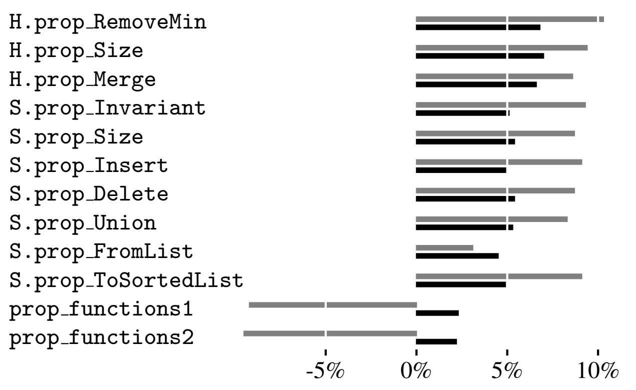


Figure 5. Relative slowdown (speedup) of example QUICKCHECK properties using `TFGen` compared to baseline runs with `StdGen` (▬). Black bars (▬) indicate how much time has been spent in the ThreeFish cipher by `TFGen`.

The proposed PRNG (Section 3), whose implementation we will refer to as `TFGen` in this section, uses a cryptographic block cipher to generate random numbers. We chose the 256-bit ThreeFish [17] block cipher, which is efficiently implementable in software. Despite the large block size, the encryption of a single block takes less time than for 128-bit AES, which is a standard contemporary block cipher. In addition, ThreeFish does not require an additional costly key setup phase, which is required by AES when a new encryption key is used. The actual implementation of the cipher used is a simplified version of the one from the Skein C reference implementation[6], which is accessed from Haskell using the Foreign Function Interface.

The encoding used in `TFGen` is the one presented in Appendix A. We chose to use 64 bits for the bit sequence in each block (bseq_i) and another 64 bits for the counter (ctr_i), leaving 128 bits unused. Choosing 64 bits for the bit sequence means that rehash is needed every 64 splits, which brings the cost of doing that to below 2%[7] in split-intensive benchmarks. Similarly, overflow of a 64 bit counter will happen very rarely and have a negligible impact on performance. Using more bits for representing the bit sequence or the counter would, on the other hand, likely cause more overhead than give benefits.

One difference between the benchmarked code and the encoding from Appendix A is that if `next` is called repeatedly, it returns subsequent words from a single generated block, generating a new block every 8 calls. Implementing this particular feature requires only small changes to the encoding.

6.1 QuickCheck

The primary application that we considered for the proposed PRNG is random testing tool QUICKCHECK. Currently, QUICKCHECK uses the default splittable PRNG in Haskell, `StdGen`, for generating random test data. QUICKCHECK's random data generators make heavy use of splitting, in order to avoid generating parts of values that are never inspected.

Figure 5 shows the relative performance of some typical properties that test implementations of an ordered set (`S.*`) and heap (`H.*`) data structures, taken from the `examples` folder in the QUICKCHECK distribution. In addition, two last properties used ran-

[6] `http://www.schneier.com/skein.html`

[7] All measurements were performed using GHC 7.6.2 targeting x86-64 architecture on Intel XEON E5620 processor (Westmere-EP) clocked at 2.4 GHz.

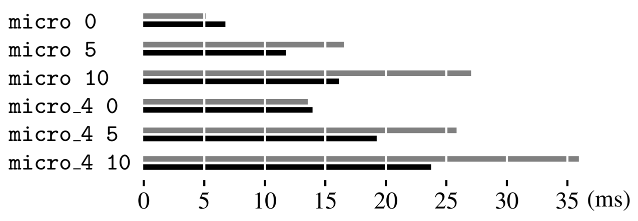


Figure 6. Run times of micro benchmarks for `StdGen` (▬) and `TFGen` (▬). Benchmark `micro` n executes 20k `next` operations and $(n + 1)$20k `split` operations. Benchmark `micro_4` is the same, except that it runs 4 `next` operations in sequence instead of one in each case.

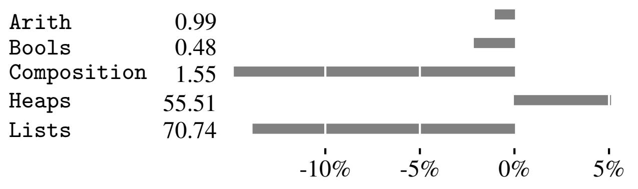


Figure 7. Run times of QUICKSPEC (in s) on its example problems with `StdGen` and relative `TFGen` performance (▬).

domly generated functions, whose generators are also provided in QUICKCHECK. The set and heap properties execute 3–11% slower (8% slower on average) with `TFGen`, compared to `StdGen`. The two last properties, on the other hand, execute about 9% quicker.

The likely explanation for this is that QUICKCHECK's random function generators perform a large number of splits. As micro benchmarks presented in Figure 6 suggest, `split` is executed 2.3x faster by `TFGen` than by `StdGen`.

On the other hand, operation `next` is over 30% slower with `TFGen` (`micro 0`) when it is called for isolated states, and never called twice in a sequence. Such a situation occurs often in QUICKCHECK generators, which can explain the slow down in the properties. However, executing subsequent `next` operations is much less expensive in `TFGen` as it only requires reading the next word from the block, and regenerating the block once every 8 words, which is confirmed by the `micro_4` benchmark.

Benchmarks using QUICKCHECK (Figure 5) also contain an estimation of the percentage of time consumed by computing the ThreeFish block cipher. The estimation has been obtained by running the properties with a modified version of `TFGen`, which runs the block cipher four times, instead of one, and discards three of the results. As can be seen from the figure, the cost of running ThreeFish is very low for all of the properties, which indicates that the run time cannot be improved much by speeding up the cipher.

6.2 QuickSpec

QUICKSPEC [14] is a tool that discovers an equational specification of Haskell code based on the behaviour it observes through random testing. Testing is performed with the use of QUICKCHECK's random data generators, and usually consumes a significant portion of total run time of QUICKSPEC.

Figure 7 shows run times of QUICKSPEC version 0.9 on examples from its `examples` folder. As seen in the figure, `Arith` and `Bools` perform the same with both generators, `Lists` and `Composition` are about 13–15% faster with `TFGen`, and `Heaps` is are about 5% slower with it. QUICKSPEC relies on generating random valuation functions for testing, which require executing many `split` operations. This is likely to have equalised the advantage `StdGen` has over `TFGen`. In addition, `Composition` and

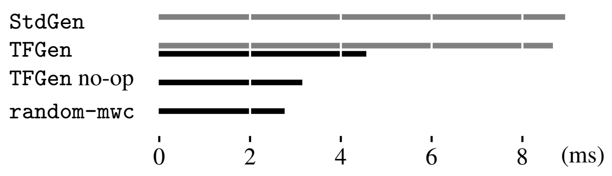


Figure 8. Run time of generating a vector of 100k 32-bit integers (in ms) using `next` (▬), and functions native for each generator (▬).

`Lists` examples contain higher-order functions in their signatures, necessitating random generation of even more functions.

6.3 Linear generation

The standard `StdGen` generator is considered to be very slow for linear random number generation. Its slowness to a large extent comes from using standard `Random` instances, which serve as high-level primitives for generating random values of different types using the numbers returned by the random generator. On top of that, code using `StdGen` usually uses the Haskell lazy list as the intermediate data structure, which adds additional overhead.

To benchmark linear generation we decided to sidestep the `Random` instances and generate numbers directly using `next`. To remove the overhead caused by using lists we used the `vector`[8] package, which allows for efficient generation of unboxed vectors.

Figure 8 shows run times for generating an unboxed vector of 100k 32-bit random numbers. The grey (▬) bars show results for code that used the `next` method from `RandomGen` class, implemented by both generators. `StdGen` returns `Int` values from range 0–2147483562. `TFGen`, on the other hand, generates the full range of `Word32` values, which are then transformed into `Int` values from `StdGen`'s range, in order to be able to benchmark it with `Random` instances, which were written for `StdGen`. Results shown in black (▬) were obtained using code that directly generated `Word32` values.

As shown in the figure, `TFGen` is marginally faster than `StdGen` at generating numbers using `next`. However, `TFGen` performs almost twice as fast when directly generating `Word32` values. The observed difference is most likely due to the code that transforms the results into the smaller range, which is simple, and yet appears to prevent some optimisations to be performed by the compiler, such as *fusion* [27]. We did not see the need to improve that code, as its only function was benchmarking that also used the `Random` instances, which themselves impose considerable overhead. However, it suggests that much performance is to be gained by giving that code, and the `Random` instances, some attention.

We chose the `random-mwc` package, which is considered to be the fastest linear PRNG for Haskell, as the baseline for comparing the 'raw' random generation speed. As shown in the figure, `TFGen` is slower by 65% than `random-mwc` in generating 32-bit random numbers. The fastest reported performance for `random-mwc` is still higher than the one measured by us, at 16.7 ns per 32-bit number (with an unreported CPU)[9], which would correspond to 1.67 ms in the figure.

The entry marked '`TFGen` no-op' in the figure is the run time of the `TFGen` generator with the code running the actual cipher replaced by a very cheap operation (XOR). This version of `TFGen` is only slower by 1.41 ms, suggesting that the generator's performance is heavily affected by book-keeping computations.

6.4 Conclusion

We found that an implementation of a high-quality splittable PRNG based on a cryptographic block cipher can have competitive per-

[8] `http://hackage.haskell.org/package/vector`

[9] `http://www.serpentine.com/blog/2011/03/18/a-little-care`

formance with respect to traditional PRNGs. We found our implementation (`TFGen`) to be, on average, 8% slower than `StdGen` on typical QUICKCHECK properties and about 9% faster on properties involving random functions, and we observed a similar speedup with QUICKSPEC. We found that linear random number generation with `TFGen` is slower by 65% than with `random-mwc`, a state-of-the-art linear random number generator for Haskell.

Measurements performed by us suggest that it will be possible to improve the performance of `TFGen` in the future by optimising the `Random` instances and making sure that the generator's code gets properly optimised together with the code that uses it.

7. Discussion and future work

Splittable* PRNG*s are keyed hash functions We showed that a splittable PRNG can be constructed using a keyed hash function in a straightforward way. The crucial observation that allowed this was that the property that is expected from keyed hash functions, that they are indistinguishable from random mappings to computationally-bounded programs, is exactly the property we need for splittable PRNGs. Similarly, a keyed hash function can be constructed based on a splittable PRNG with a reasonable efficiency by mapping each possible input to the hash function to a unique sequence of PRNG operations. In that case, the security of the keyed hash function would depend on the pseudorandomness of the PRNG. Based on this observation, both these constructions appear to *solve the same problem.*

Bounds As shown in Section 4, the presented splittable PRNG construction has a bound on the order of $2^{-b}(q^3 l + q^2 l^2 + tql)$. In contrast, the bound for a linear PRNG based on a block cipher running in the CTR mode is $2^{-b}(q^2 + tq)$ [33], which can be shown using the Switching Lemma, mentioned in Section 4.3. Thus, by using a splittable PRNG we have to trade a worse bound for the flexibility that splitting provides. The bound on the splittable PRNG used as a linear generator trivialises to $2^{-b}(q^3 + tq)$, which is a worse result than the bound derived directly. It may be possible to improve our analysis and get a better bound for the generator.

Analysis performed in [5] for the Merkle-Damgård construction shows that the best bound that can be achieved with this construction is on the order of $2^{-b} l q^2$, if the compression function is modelled as a pseudorandom function. However, other similar constructions can provide better bounds (see below).

Alternative hashing constructions Thanks to basing a splittable PRNG on hashing, any keyed hash function can be used for its construction. Keyed hash functions are commonly used as Message Authentication Codes (MACs), and there is a large variety of constructions available.

One construction that we considered is CBC-MAC [7], which is a standard way of creating a MAC on top of a block cipher. Its mode of operation is very similar to the construction used by us, and it also requires a prefix-free set of inputs. The main difference from our construction is that the intermediate state of CBC-MAC contains both a block with the result of previous block cipher run, and the key, which needs to be kept during the entire computation. In practical terms, this would mean that the state of the generator would have to keep one more pointer to the key that would be shared by all states. On the other hand, a better bound has been proved for CBC-MAC than for our construction, namely on the order of $2^{-b} l q^2$ [7] (when $l < 2^{b/3}$). Therefore, the trade-off between slightly slower performance and better bound should be explored.

A variation of this construction, called ECBC-MAC, has even better bound $2^{-b} l^{o(1)} q^2$, where $l < 2^{b/4}$ and $o(1)$ is a diminishing function. However, ECBC-MAC requires an extra encryption step at the end of hashing, which would impose large overhead in a splittable PRNG.

Prefix-freedom Another design choice that we considered was whether to use a hash function, which requires the set of its inputs queried in one run to be prefix-free, or a hash function that does not have this requirement. Hash functions that do not require this are, in general, less efficient (for example ECBC-MAC), and usually need to perform more work at the end of hashing, which makes them less suitable for our application.

A related issue concerns the API requirement that `next` and `split` cannot be called on the same state. This requirement results in the set of paths to states queried in one program run to be prefix-free. Relaxing this requirement would be possible, but would require another, possibly less efficient encoding. However, we found that this API requirement is reasonable for the reason of compositionality, not performance. Consider a composable random data generator [13] that generates random lists given a generator for their elements, and another generator for integers:

```
myList :: Gen a -> Gen [a]
myInt  :: Gen Int
```

Generator `myList` will internally perform some number of `split` operations, which may depend on random numbers that it had consumed itself. Then it will use its argument generator to generate elements of the list, giving a different generator state to it each time. Now consider that the API requirement has been dropped, and that of two different states given to the argument generator, one may be derived from the other. If the argument generator also performs the `split` operation, as would be the case when we call `myList (myList myInt)`, then two equal states may be used in different places of the program, leading to the same numbers being generated.

Thus, to be safe `myList` must make sure that it does not call the argument generator with states that may have been derived from each other. The easiest way to ensure that is to never call `split` on a state which is passed to a subcomputation, which is essentially the strategy used for satisfying the original API requirement. It is thus likely that compositional use of the API was the reason for this requirement to be created.

8. Related work

Splittable PRNGs based on a Linear Congruential Generator (LCG) proposed by Fredrickson et al. [19] ensured that a number of right-sequences of bounded length starting from a single left-sequence are disjoint. Mascagni et al. discuss a number of traditional PRNGs (LCGs and others) that can be used as parallel generators. However, none of these two works supports unlimited on-demand splitting.

Burton and Page [10] discuss a number of splittable PRNGs for Haskell, based on an LCG. The ideas include (1) distributing halves of the random sequence, which would exhaust the sequence very fast, (2) using a non-deterministic solution, and (3) randomly jumping in the sequence on split. The last solution looks promising, but the only statistical argument presented in favour of it is the measured lack of local state collisions. We implemented the last solution and found it to be slower than our generator.

The idea about using a block cipher to implement a splittable PRNG has appeared on the Haskell-Cafe mailing list [32]. The proposed design keeps a block cipher's key and a counter and implements `split` as follows, by regenerating the key in the right derived state at `split` operation.

$$\texttt{split}\ (k, n) = ((k, n+1), (\mathsf{enc}_k\ (n), 0))$$

The rationale behind it is that the randomness of block cipher encryption will carry over to the whole construction. However, it is not formally justified why it is the case, or how many splits can be executed without compromising the randomness. In our view the scheme *is* correct, since in fact it is an instance of the generator proposed by us, using a suitable encoding, and thus is covered by our correctness argument. Its disadvantage is the high cost of `split` due to having to run the block cipher for each right derived generator state.

Random number generators specified in NIST SP 800-90A [2] are designed to provide unpredictable random input for cryptographic applications. The generators have pseudorandom cores (DRBGs[10]), each of which is based on a different cryptographic primitive, such as a block cipher, a keyed hash function (HMAC), a non-keyed hash function or Eliptic Curves (EC).

The generators can be seeded using other generators' pseudorandom output, which been used for implementing splitting in the `crypto-api` Hackage package.[11] Unfortunately, while all the generators appear to be correct, the NIST publication does not formalise or prove any aspects of any of them. The pseudorandom parts of the HMAC and EC generators have been analysed elsewhere [9, 23], however the proofs do not cover seeding one generator using another generator's output. Furthermore, purely deterministic (up to initial seeding) functionality is likely not to be the main focus of these generators, as cryptographic applications require frequent reseeding with external entropy [2].

Micali and Schnorr [31] present a PRNG based on the `rsa` cryptosystem, which is provably random and supports n-way splitting. However, the randomness proofs are asymptotic, which means that they do not indicate what parameters to choose to achieve a particular level of indistinguishability, or whether the generator can practically achieve reasonable randomness [12, 35].

Leiserson et al. [26] propose a PRNG for deterministic parallel random number generation. Their generator is integrated with the MIT Cilk platform, which tracks the call path of the current program location, which is then used to generate random numbers in a way that is independent of thread scheduling. To generate random numbers, the generator hashes the call path using a specially-constructed hash function. The hash function first compresses the path minimising the likelihood of collisions, and then applies a mixing operation. The compressing function ensures that the probability of collisions is low, while the mixing function provides randomisation. It is only the former of these properties that is formally stated and proved. The quality of the generated random numbers obviously depends on the quality of the mixing function, but it is hard to say what level of randomness it provides, especially that the presented results of statistical tests include failures.

The generator supports paths of length up to 100 (or similar value), due to the fact that the whole path must hashed when a random number is requested, and that a vector of random 'seed' values of the same length as the path is required. Thus, the general construction could not be considered to use bounded space, although the paper considers adapting it into an incremental one, as well as using alternative methods for path compression.

Salmon et al. [34] present a high-performance PRNG based on a block cipher running in CTR mode. Their generator is parallelisable, but does not support splitting on demand. Their proposed generator solves two problems of traditional PRNGs, namely that they are difficult to parallelise and that their quality is unproven, and often low. The proposed generator is correct, but the randomness claims are only stated informally. The authors consider a number of block ciphers, such as ThreeFish and AES, but also their weakened versions, which offer higher performance. Finally, they propose a parallel random number generation API, which separates keeping track of counters from random number generation. The generator does not provide functionality equivalent to on-demand splitting, as it is the application that is responsible for 'distributing' the independent random streams.

[10] Deterministic Random Bit Generators

[11] `http://hackage.haskell.org/packages/archive/crypto-api/0.12.2.1/doc/html/Crypto-Random.html`

9. Conclusion

In this paper, we show that cryptographic keyed hash functions are attractive means for implementing splittable PRNGs. While the general construction of a splittable PRNG shown by us can be based on any keyed hash function, we propose using a well-known and efficient keyed hash function based on a cryptographic block cipher.

The hash function itself is based on a provably secure construction, which is guaranteed to yield high-quality randomness under the assumption that a secure block cipher is used. Our Haskell implementation is only marginally slower than Haskell's default splittable PRNG, which makes it a promising drop-in replacement.

The proposed design also suggests a new operation `splitn`, which would speed up some of the split-intensive code, and could be added to the API of splittable PRNGs in Haskell.

Acknowledgments

This research has been supported by the Resource Aware Functional Programming (RAW FP) grant awarded by the Swedish Foundation for Strategic Research.

References

[1] L. Augustsson, M. Rittri, and D. Synek. Functional pearl: On generating unique names. *J. Funct. Program.*, 4:117–123, 1 1994.

[2] E. Barker and J. Kelsey. NIST Special Publication 800-90a: Recommendation for random number generation using deterministic random bit generators, 2012.

[3] M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. `http://eprint.iacr.org/2004/331`.

[4] M. Bellare and P. Rogaway. Introduction to modern cryptography, 2005. `http://www.cs.ucsd.edu/~mihir/cse207/classnotes.html`.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: the cascade construction and its concrete security. In *Foundations of Computer Science*, pages 514–523, 1996.

[6] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proc. Foundations of Computer Science, 1997*, pages 394–403, 1998.

[7] M. Bellare, K. Pietrzak, and P. Rogaway. Improved security analyses for CBC MACs. In *Advances in Cryptology — CRYPTO 2005, LNCS 3621*, pages 527–545. Springer-Verlag, 2005.

[8] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, Nov. 1984.

[9] D. R. L. Brown and K. Gjøsteen. A security analysis of the NIST SP 800-90 elliptic curve random number generator. In *Proc. Advances in cryptology — CRYPTO '07*, pages 466–481. Springer-Verlag, 2007.

[10] F. W. Burton and R. L. Page. Distributed random number generation. *J. Funct. Program.*, 2(2):203–212, 1992.

[11] S.-j. Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition.* NIST, 2012.

[12] S. Chatterjee, A. Menezes, and P. Sarkar. Another look at tightness. In *Proc. Selected Areas in Cryptography (SAC'11), LNCS. 7118*, pages 293–319, 2012.

[13] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. International Conference on Functional programming*, ICFP '00, pages 268–279. ACM, 2000.

[14] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *Proc. Tests and Proofs*, TAP'10, pages 6–21. Springer-Verlag, 2010.

[15] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-damgård revisited: How to construct a hash function. In *Advances in Cryptology — CRYPTO 2005, LNCS 3621*, pages 430–448. Springer-Verlag, 2005.

[16] I. Damgård. A design principle for hash functions. In *Advances in Cryptology — CRYPTO '89, LNCS 435*, pages 416–427. Springer, 1990.

[17] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family, 2010. URL `http://www.schneier.com/skein.pdf`.

[18] R. Fischlin and C. Schnorr. Stronger security proofs for rsa and rabin bits. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97, LNCS 1233*, pages 267–279. Springer, 1997.

[19] P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, and T. Warnock. Pseudo-random trees in monte carlo. *Parallel Computing*, 1(2):175–180, 1984.

[20] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, Aug. 1986.

[21] C. Hall, D. Wagner, J. Kelsey, and B. Schneier. Building PRFs from PRPs. In *Advances in Cryptology — CRYPTO '98, LNCS 1462*, pages 370–389. Springer-Verlag, 1998.

[22] D. R. C. Hill, C. Mazel, J. Passerat-Palmbach, and M. K. Traore. Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, 2012.

[23] S. Hirose. Security analysis of DRBG using HMAC in NIST SP 800-90. In K.-I. Chung, K. Sohn, and M. Yung, editors, *Information Security Applications*, pages 278–291. Springer-Verlag, 2009.

[24] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any One-way function. *SIAM Journal on Computing*, 28:12–24, 1999.

[25] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), 2007.

[26] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc. Symp. on Principles and Practice of Parallel Programming*, pages 193–204. ACM, 2012.

[27] R. Leshchinskiy. Recycle your arrays! In *Proc. Practical Aspects of Declarative Languages*, PADL '09, pages 209–223. Springer-Verlag, 2009.

[28] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, 2000.

[29] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson. Recent developments in parallel pseudorandom number generation. In *SIAM Conf. on Parallel Processing for Scientific Computing*, volume II, pages 524–529, 1993.

[30] B. D. McCullough. The accuracy of econometric software. In D. A. Belsley and E. J. Kontoghiorghes, editors, *Handbook of Computational Econometrics*, chapter 2, pages 55–79. Wiley, 2009.

[31] S. Micali and C. P. Schnorr. Efficient, perfect polynomial random number generators. *J. Cryptology*, 3:157–172, 1991.

[32] S. Peyton-Jones, B. Smith, et al. Splittable random numbers. Mailing list discussion, 2010. URL `http://www.haskell.org/pipermail/haskell-cafe/2010-November/085959.html`.

[33] P. Rogaway. Evaluation of some blockcipher modes of operation. Unpublished manuscript, 2011.

[34] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proc. High Performance Computing, Networking, Storage and Analysis*, pages 1–12. ACM, 2011.

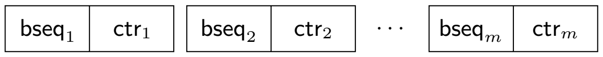
[35] A. Sidorenko and B. Schoenmakers. Concrete security of the Blum-Blum-Shub pseudorandom generator. In *Cryptography and Coding 2005, LNCS 3796*, pages 355–375, 2005.

[36] A. C. Yao. Theory and application of trapdoor functions. In *Proc. Symp. Foundations of Computer Science*, pages 80–91. IEEE, 1982.

A. Appendix. Efficient encoding

We present an efficient encoding of paths containing the `next` operation. The encoding uses the general idea of keeping the bit sequence and counter separate, as does `e1`.

We use the following scheme (`e4`) to encode the bit sequence and the counter in a sequence of blocks. Each block has a fixed region for encoding part of the bit sequence and another one for encoding the counter. The following diagram shows the data layout of a sequence of blocks that encodes a path.



Regions marked ctr_i contain binary numbers ranging from 0 to N. Regions marked bseq_i contain B-bit segments of a bit sequence, except the last segment, which may be shorter. The main part of the encoding is defined inductively, as a function `e3` mapping a path into a sequence of pairs of numbers and sequence of bits, each pair representing a block. The following definition assumes that $\mathsf{e3}\ (c) = \langle \dots, (b, n) \rangle$ for recursive cases.

$$\mathsf{e3}\ (\langle\rangle) = (\langle\rangle, 0) \tag{A.3a}$$

$$\mathsf{e3}\ (c \| \langle \texttt{left} \rangle) = \begin{cases} \langle \dots, (b \| \langle 0 \rangle, n) \rangle & \text{if } |b| < B, \\ \langle \dots, (b, n), (\langle 0 \rangle, 0) \rangle & \text{otherwise.} \end{cases} \tag{A.3b}$$

$$\mathsf{e3}\ (c \| \langle \texttt{right} \rangle) = \begin{cases} \langle \dots, (b \| \langle 1 \rangle, n) \rangle & \text{if } |b| < B, \\ \langle \dots, (b, n), (\langle 1 \rangle, 0) \rangle & \text{otherwise.} \end{cases} \tag{A.3c}$$

$$\mathsf{e3}\ (c \| \langle \texttt{next} \rangle) = \begin{cases} \langle \dots, (b, n+1) \rangle & \text{if } n < N', \\ \langle \dots, (b \| \langle 1 \rangle, 0) \rangle & \text{if } n = N' \text{ and } |b| < B, \\ \langle \dots, (b, N), (\langle\rangle, 0) \rangle & \text{otherwise.} \end{cases} \tag{A.3d}$$

Adding an operation at the end of the path changes only the last block of the resulting sequence and possibly adds another one after it. Operations `left` and `right` add one bit to the segment in the last block. If the segment is full, a new block is started. Operation `next` increments the counter in the last block. Valid values of the counter are $0 \dots N' = N - 1$. In the event of an overflow, bit '1' is added to the last segment. If the segment is full, special value N is used as the counter and a new block is added.

The result of the function is transformed into actual sequence of blocks by encoding the numbers in binary and putting the segments verbatim. The last incomplete segment is zero-padded. Let to_block be the function that turns a pair into a block. We omit it's definition, but we note the important property that we require from it.

Proposition A.1. *Let (b_1, n_1) and (b_2, n_2) be inputs to to_block function. If $n_1 \neq n_2$ or $b_1 \prec b_2$ and $b_2 \neq b_1 \| \langle 0, \dots, 0 \rangle$*[12] *or $b_1 \neq b_2$ then to_block $(b_1, n_1) \neq$ to_block (b_2, n_2).*

Note that linear random number generation using `next` requires only one iteration of the hash function for most random numbers as only the counter in the last block is changing. This way, when `next` is called repeatedly, the block cipher used as the compression function is effectively run in counter mode for generating random numbers. Similarly, splitting usually requires updating only the last block.

Proofs about the encoding have been omitted for the lack of space.

[12] We use the shorthand $b_2 \neq b_1 \| \langle 0, \dots, 0 \rangle$ to denote that b_2 is not b_1 extended with some number of zeroes.