On Seedless PRNGs and Premature Next

Sandro Coretti
IOHK, Zürich, Switzerland

Yevgeniy Dodis
New York University, NY, USA

Harish Karthikeyan
New York University, NY, USA

Noah Stephens-Davidowitz
Cornell University, Ithaca, NY, USA

Stefano Tessaro
University of Washington, Seattle, WA, USA

Abstract

Pseudorandom number generators with input (PRNGs) are cryptographic algorithms that generate pseudorandom bits from accumulated entropic inputs (e.g., keystrokes, interrupt timings, etc.). This paper studies in particular PRNGs that are secure against *premature next* attacks (Kelsey et al., FSE '98), a class of attacks leveraging the fact that a PRNG may produce an output (which could be seen by an adversary!) *before* enough entropy has been accumulated. Practical designs adopt either unsound entropy-estimation methods to prevent such attacks (as in Linux's `/dev/random`) or sophisticated pool-based approaches as in Yarrow (MacOS/FreeBSD) and Fortuna (Windows).

The only prior theoretical study of premature next attacks (Dodis et al., Algorithmica '17) considers either a *seeded* setting or assumes constant entropy rate, and thus falls short of providing and validating practical designs. Assuming the availability of random seed is particularly problematic, first because this requires us to somehow generate a random seed without using our PRNG, but also because we must ensure that the entropy inputs to the PRNG remain independent of the seed. Indeed, all practical designs are seedless. However, prior works on seedless PRNGs (Coretti et al., CRYPTO '19; Dodis et al., ITC '21, CRYPTO'21) do not consider premature next attacks.

The main goal of this paper is to investigate the feasibility of theoretically sound seedless PRNGs that are secure against premature next attacks. To this end, we make the following contributions:

1. We prove that it is impossible to achieve seedless PRNGs that are secure against premature-next attacks, even in a rather weak model. Namely, the impossibility holds even when the entropic inputs to the PRNG are independent. In particular, our impossibility result holds in settings where seedless PRNGs are otherwise possible.
2. Given the above impossibility result, we investigate whether existing seedless pool-based approaches meant to overcome premature next attacks in practical designs provide meaningful guarantees in certain settings. Specifically, we show the following.
   - We introduce a natural condition on the entropic input and prove that it implies security of the round-robin entropy accumulation PRNG used by Windows 10, called Fortuna. Intuitively, our condition requires the input entropy "not to vary too wildly" within a given round-robin round.
   - We prove that the "root pool" approach (also used in Windows 10) is secure for general entropy inputs, provided that the system's state is not compromised *after* system startup.

2012 ACM Subject Classification Security and privacy → Mathematical foundations of cryptography; Security and privacy → Information-theoretic techniques; Theory of computation → Pseudorandomness and derandomization

Keywords and phrases seedless PRNGs, pseudorandom number generators, PRNG, Fortuna, premature next

Digital Object Identifier 10.4230/LIPIcs.ITC.2022.9

Funding *Yevgeniy Dodis*: Partially supported by gifts from VMware Labs and Google, and NSF grants 1815546 and 2055578.
Stefano Tessaro: Supported in part by NSF grants CNS-1930117 (CAREER), CNS-1926324, CNS-2026774, a Sloan Research Fellowship, and a JP Morgan Faculty Award.

1 Introduction

Pseudo-random number generators (PRNGs) are one of the most critical building blocks of secure systems. In particular, no meaningful cryptography is achievable without (pseudo) randomness. In practice, PRNGs' main functionality is to *accumulate* entropy (modeled by the function refresh in the syntax) into one or more pools from several sources (such as keystrokes, interrupt timings, etc.), and to then *extract* "clean" pseudorandom bits from these pools (modeled by the function next). In other words, refresh calls is used to accumulate entropy into the state of the PRNG while next is used to produce outputs from this PRNG state. While doing this, PRNGs must resist powerful attacks. On the one hand, the available entropy sources (i.e., the input to the PRNG) may be partially controlled by the adversary interacting with the system. On the other hand, the state of the PRNG may be compromised, and we want to protect both prior uses of the PRNG (i.e., we want forward security), as well as allow for recovery from such compromise. PRNGs, in particular, differ from "traditional" pseudorandom generators, which instead already assume a fully entropic input.

Several practical PRNG designs have been proposed, including those in operating systems such as `/dev/random` [20] for Linux, Yarrow [15] for MacOS/iOS/FreeBSD, and Fortuna [10] for Windows, and in standards like NIST's SP 800-90A [2]. Designing secure PRNGs remains however a complex task, and several flaws have been identified in existing designs (cf. e.g. [19, 21]).

Seedless PRNGs. One could hope that the best way forward is to develop provably secure PRNGs, following a line of work initiated by Barak and Halevi [1]. Yet, theoretical validation presents several technical challenges. In particular, we want such PRNGs to be *general*, in that they achieve security under the minimal assumption that the available sources have sufficient entropy. To address this, much of the prior work has considered a *seeded* setting, first proposed by Dodis et al. [7]. Here, the PRNG can rely on a random *seed* which is *independent* from the accumulated entropy (but known to the attacker), an approach inherited from the necessity of seed for extraction from general entropy sources [17]. Such a seeded approach was taken by several subsequent works [7, 8, 19, 11, 14, 12]. This seed serves as an input to both refresh and next.

However, the seeded setting is not necessarily practical. Indeed, since our end goal is that of generating randomness in the first place, one may question where a uniformly random and independent seed would come from! Moreover, and perhaps even more importantly, it is unreasonable to expect our input sources to be truly independent of the seed. (E.g., our future keystrokes can certainly depend on the prior output of our PRNG, which depends on the seed.) Unsurprisingly, all practical designs are seedless.

This issue has motivated recent work studying different ways in which the impossibility of deterministic extraction can be circumvented without the need for seed. Coretti et al. [4] consider constructions based on cryptographic hash functions modeled as random oracles and introduced corresponding meaningful notions of entropy in this setting. The formal definition is presented as Definition 1. Subsequent works by Dodis et al. [6, 5] consider the simple case in which the inputs are *independent* and without assuming ideal primitives.

This paper: Seedless PRNGs & Premature-next attacks. All prior theoretical work on *seedless* PRNGs relied heavily on the assumption that the PRNG is allowed sufficient time to accumulate entropy before having to provide any output, i.e., they do not handle so-called *premature-next* attacks [16]. In such an attack, the adversary requests output from the PRNG before it has accumulated enough entropy to guarantee security. Much prior work (including all prior work in the seedless setting) simply assumes that all accumulated entropy is lost upon such a premature-next call. With such a definition, a PRNG might fail to produce a single pseudorandom bit, regardless of how much entropy is provided!

Linux's `/dev/random` [20] attempts to overcome premature-next attacks by blocking the RNG as long as insufficient entropy has been accumulated, but this approach cannot be theoretically sound, as estimating entropy is impossible [18, 10]. Indeed, a concrete way to fool `/dev/random`'s entropy estimation was given in [7]. In contrast, Yarrow [15] and Fortuna [10] propose a clever solution to the problem. Abstractly, these constructions have a *register* as well as many *pools*. Only the register is used to provide output. Each time these PRNGs receive some input, they "add it to one pool" selected in a round-robin fashion. Then, at different rates, each pool is used to occasionally update the register. We call this "emptying the pool." Intuitively, while a premature-next call might completely leak the input to any pools that have been emptied recently, it will not reveal any information about inputs to pools that have not been emptied since they received this input.

A generalization of this pool-based approach was analyzed formally by Dodis et al. [8]. However, their analysis either assumes seeds (thus departing from the deterministic approach taken by Yarrow/Fortuna) *or* requires that the entropy rate is constant – i.e., that all inputs have the same (unknown, adversarially chosen) entropy. Both situations are undesirable, and in this paper, we aim to make progress on the following general question:

Can we have seedless PRNG designs which provably resist (in some meaningful way) premature-next attacks?

1.1 Our Results

Impossibility of seedless PRNGs. We first address the feasibility question of whether seedless PRNGs can, in principle, be secure against premature-next attacks. We would like in particular to assess whether recent positive results on seedless PRNGs, [4, 6, 5] can be extended to resist premature-next attacks.

Notice that, if the attacker can choose to vary the entropy of the inputs, then no "deterministic pool-based approach" can work. (As in [8], we formalize this below using the notion of a scheduler.) In particular, if we require γ bits of entropy to go into a single pool in order to recover from compromise and the attacker knows when pools will be filled and emptied, then the attacker can simply provide a bit less than γ bits to each pool before it is emptied. (This intuition is formalized in [8].) However, one can imagine much more complicated constructions. E.g., we might choose which pool to fill or empty based on the (entropic) input (perhaps even with some attempt at entropy estimation like `/dev/random`), or we might not use a pool-based approach at all.

Surprisingly, we show that *no* seedless PRNG can resist premature next attacks, even if the inputs are sampled independently. In particular, as deterministic extraction without the seed *is* possible for independent inputs [3], our impossibility is inherently due to the premature next problem. In more detail, following [8], we parameterize security by two values, γ^*, and β. The goal is to guarantee that *if* the PRNG has obtained γ^* bits of min-entropy within T^* steps after the last state compromise, then the PRNG will revert to producing pseudorandom bits within βT^* steps after the same state compromise. We prove that for any choice of γ^*, β, there exists an efficient adversary providing $q \geq {\gamma^*}^2\beta^2$ PRNG inputs (each with one independent bit of entropy) which violate the PRNG security against premature next attacks. Since q is typically huge, this rules out any reasonable settings of γ^* and β.

In addition to being interesting in its own right, this shows a natural setting where meaningful PRNG security (e.g., entropy accumulation and extraction) is possible *without* premature-next attacks, but impossible *with* them.

Toward Positive Results. The above strong impossibility result, including the "separation" between randomness accumulation/extraction and premature-next security, motivates us to search for positive results even (optimistically) assuming *perfect entropy accumulation and extraction.* In fact, we already have two widely used solutions that appear to work in practice. First, we already mentioned the round-robin pool-based approach, called Fortuna, which is part of Windows 10 and macOS. Second, Windows 10 [9] uses a special "root pool" to solve the problem of initial entropy accumulation when the computer starts up. This single pool is emptied at exponentially increasing intervals (e.g., at time $1, \beta, \beta^2, ...$) to (heuristically) solve the problem that sometimes the computer might boot with no good source of randomness for an unknown period of time. Intuitively, if good entropy starts to come in at (unknown) time t, the root pool will allow the PRNG to produce good random bits by time at most βt. While this simple approach does not work when one is worried about state compromise at an unknown time (and this is why more than 1 pool is used for general purpose PRNGs like Fortuna), it appears quite effective for accumulating entropy at startup.

Given the existence of these two heuristics to accumulate entropy within pools, we ask whether we can find natural conditions where these approaches *provably* work, despite our strong impossibility result above. To make this question formal, we define a clean model of seedless (pool-based) *schedulers*, extending the corresponding notion of schedulers [8] to the seedless setting. Intuitively, if we have k pools, given each entropic sample X_i, the scheduler decides which pool $\mathsf{in}_i \in [k]$ will accumulate this entropy, and, which pool $\mathsf{out}_i \in [k]$ (if any) will contribute its accumulated entropy back to the register. Moreover, to model ideal entropy accumulation and extraction, we assume that the entropy that was thrown to pool i simply adds up without loss.

In fact, at this level of abstraction, we can completely forget about entropy and PRNGs and simply consider an abstract notion of a scheduler, whose goal is to distribute a sequence of weights $w_1, \ldots, w_q \in [0, 1]$ into pools, sometimes emptying one of the pools with the following guarantee. If there are t consecutive weights $w_{t_0+1}, \ldots, w_{t_0+t}$ whose sum is larger than some threshold α, then there should be a pool that accumulates at least weight 1 in this same time period (without being emptied) and is emptied shortly thereafter, say before time step $t_0 + \beta t$. We call this (α, β)-security. Here, a pool accumulating weight 1 in this abstract scheduler game corresponds to a pool accumulating sufficient entropy in a pool-based PRNG. [8] proved formally that a secure scheduler can be used to convert PRNGs that are secure in a model that does not allow for premature-next attacks (used for the individual entropy pools) into a PRNG that is secure in a model with premature-next attacks. In particular, given an (α, β)-secure scheduler together with a PRNG that recovers from compromise after receiving γ bits of entropy *without allowing for premature-next attacks*, we can construct a PRNG that recovers from compromise even in the presence of premature next in time βt, where t is the time needed to receive $\alpha\gamma$ bits of entropy.

Because of our general impossibility result above, we cannot achieve general (α, β)-security. (We also give direct proof of this fact in the setting of schedulers below.) We then show two positive results yielding proven security guarantees for the two schedulers used in the real world, by giving meaningful restrictions to the model.

- First, we show that the root-pool approach achieves nearly optimal (α, β)-security to accumulate entropy at start-up, where $\alpha \approx \log_\beta q$ (and we can take any integer $\beta \geq 2$). Plugging in known constructions yields a PRNG in the root-pool model (i.e., in which we assume that compromise only happens at time 0) that is exponentially better than our general-scheduler lower bound stating $\alpha\beta \geq \sqrt{q}$.
- Second, we show that the round-robin Fortuna construction with $k \geq \log_\beta q$ pools achieves (α, β)-security with $\alpha \approx \log_\beta q$, provided one uses a more conservative notion of entropy called k-smooth entropy.[1] For constant-rate entropy sources, this notion of entropy is identical to the traditional min-entropy, and our result indeed generalizes the earlier observation of [8] regarding constant-rate sources. More generally, our notion of k-smooth entropy essentially captures the idea that wildly fluctuating entropy should be penalized, which we believe is a practically relevant idea (and in particular seems to be behind the heuristics used in practice). In other words, despite simple attacks on the Fortuna scheduler in the unrestricted setting, we found a natural condition where this scheduler works.

We stress that our scheduler results only solve the premature next problem assuming ideal entropy accumulation and extraction, but we hope future work will extend them to full-blown PRNGs, which provably overcome our negative results under similar restrictions.

2 Preliminaries

We write $\mathbb{N} := \{0, 1, 2, \ldots\}$ for the set of natural numbers and for positive integers $k \geq 1$, we write $[k] := \{0, \ldots, k-1\}$ for the natural numbers up to $k-1$. When a value x is sampled uniformly from a distribution X, we will denote it by $x \leftarrow X$. By U_n, we will denote a uniform distribution over bit strings of length n.

We consider PPT adversaries, in some security parameter λ. All our variables in our security definitions will depend on this security parameter.

Min-Entropy. The *prediction probability* of a random variable X is $\mathsf{Pred}(X) := \max_x \mathsf{P}[X = x]$ and the *min-entropy* is $\mathrm{H}_\infty(X) = -\log(\mathsf{Pred}(X))$.

Security Games. All of the security properties considered in this paper are captured by considering a game between a challenger and an attacker $\mathcal{A}$, both of which may have access to an ideal primitive P. The goal of the attacker is to guess a random bit b chosen by the challenger, who offers a set of oracles to the attacker to aid with this task. The *advantage* of $\mathcal{A}$ is defined as

$$2 \cdot \left| \mathsf{P}[\mathcal{A} \text{ wins}] - 1/2 \right| ,$$

where the probability is over the randomness of $\mathcal{A}$, of the challenger, and of the ideal primitive. The cases where $b = 0$ and $b = 1$ are referred to as the *real world* and the *ideal world*, respectively. One may equivalently consider $\mathcal{A}$'s advantage at telling these two worlds apart, i.e.,

$$\left| \mathsf{P}[\mathcal{A} = 1 | b = 0] - \mathsf{P}[\mathcal{A} = 1 | b = 1] \right| .$$

[1] We have a general bound for all k, including a constant number of pools, where $\alpha = O(k)$ and $\beta = O(q^{1/k})$.

3 Impossibility of "Premature Next" Seedless PRNGs

This section considers the security of seedless PRNGs against *premature next attacks* [16]. The idea behind such an attack is that next – the algorithm extracting pseudorandom bits from the PRNG state – is called before the state has accumulated sufficient entropy. The resulting output will therefore not be fully random, and an adversary can potentially use the output of many such calls to recover the state. The notion of robustness against premature-next attacks was formalized by Dodis et al. [8]. Their work generalized and analyzed a key technique to mitigate such attacks that originated in the designs of the Yarrow [15] and Fortuna [10] PRNGs. Roughly, the key idea is that the entropic inputs to the PRNG are carefully distributed to several "smaller" PRNGs, which we refer to as *pools*, and, with different frequencies, these pools are used to randomize a *register* from which random bits are extracted. (We formalize this approach in detail below.) While both Yarrow and Fortuna use deterministic *scheduling* strategies to assign entropic inputs to a pool and to decide when each pool contributes to the register, the provable robustness against premature-next attacks is achieved in [8] by relying on a *random seed* (independent from the inputs) to ensure that the entropy received from the adversary is roughly evenly distributed among the pools.

It is not hard to see that the fixed pool assignment schedule adopted by Yarrow/Fortuna cannot be robust against premature next attacks without extreme restrictions on the adversaries (e.g., the constant rate restriction). However, other seedless strategies are possible (e.g., one could assign entropic inputs to pools chosen depending on the inputs themselves, or some previous inputs; or one might try to divide each input up into smaller pieces in some way; or one might not use pools at all), and the larger question remains on the feasibility of a *seedless* PRNG which is robust, even with premature next calls. One of course should exercise some care, a fully secure deterministic PRNG cannot exist (regardless of premature-next attacks) for the same reasons deterministic extraction is impossible. So, we must make some restrictions on the input distributions provided by the adversary. For this reason, in the following, we will focus on the case of *independent* inputs, for which deterministic extraction is–in principle–possible.

Even in this setting, the main result of this section is an impossibility result. (So, the fact that we restrict our attention to independent inputs simply makes our result stronger.) Specifically, we show that it is impossible to have such a seedless PRNG which is robust against premature next attacks, even in a setting where the entropic inputs are independent.

Before we present our result, which is stated below as Theorem 5, we introduce some more syntax and definitions.

3.1 Pseudorandom Number Generators with Input

In this section, we will briefly recall the syntax of this primitive. We will use the seedless definition for this paper. We refer the readers to the work of Coretti et al. [4] for a detailed exposition.

Syntax. A PRNG is a stateful cryptographic primitive that accumulates entropy by absorbing inputs which it then uses to produce pseudorandom bits when the entropy of its state is high. A PRNG consists of two algorithms as defined below:

▶ **Definition 1** (Syntax of PRNGs). *A* pseudorandom number generator with input (PRNG) *is a pair of algorithms* PRNG = (refresh, next) *sharing a μ-bit state s, where*

- refresh *takes a state* s *and an input* $x \in \{0,1\}^m$ *and produces a new state* $s' = \mathsf{refresh}(s, x)$*, and*
- next *takes a state* s *and produces a new state and an output* $y \in \{0,1\}^r$*, i.e,* $(s', y) = \mathsf{next}(s)$*.*

A PRNG processing m*-bit inputs and producing* r*-bit output is called a* (m, r)-PRNG*.*

For our impossibility result, we will focus on $(1,1)$-PRNG. This is without loss of generality, as we could always buffer m such entropic inputs before applying a "bigger" refresh call on m such bits, and impossibility for 1 output bit implies that for $r \geq 1$ output bits.

Security. The work of Coretti et al. [4] dealt with robustness security game, *without* support for Premature Next (ROB). For purposes of this paper, we will focus on robustness security *with* Premature Next (NROB), as defined in Figure 1. While we adapt the original definition from [8] to the seedless setting, we note that we present a highly simplified security game that is enough to provide for our impossibility result. (We also leave out some functionality that is not necessary for the attacker in our impossibility result, which again simply makes our impossibility result much stronger.)

Most significantly, we assume that all of the samples provided by the attacker are *independent* from each other (which makes out impossibility much result stronger). Formally, attacker outputs a distribution X_i for the next entropic sample, and the security game independently samples a concrete value $x_i \leftarrow X_i$ from this distribution, without giving any side information back to the attacker. This allows for much simpler accounting for entropy, – by simply adding individual entropy of samples X_i produced by the attacker, – without worrying about (quite subtle) conditional entropy of such samples.

In more detail, NROB game allows adversary $\mathcal{A}$, whose state is represented by the variable σ, to access the following oracles:

- **get-next** allows the attacker to get pseudorandom outputs by calling the next procedure on the current state and returning the output y.
- **next-ror** creates a challenge, i.e., if $b = 1$, it outputs a uniform random value $y_1 \in \{0,1\}$ instead of the PRNG output y_0. Here, the PRNG output is second part of the output of next procedure.
- **get-state** models state compromises by revealing the value of the state of the adversary.

▶ **Definition 2** (Definition of an Attacker)**.** *An attacker* $\mathcal{A}$ *is called a* (q, τ)*-attacker if it provides at most* q *input distributions for* refresh *and runs in time at most* τ*.*

For security, the game keeps track of the entropy counter c which counts the entropy the attacker injected into the system since the latest compromise. When c reaches a critical value γ^*, we would like our PRNG to recover. However, instead of demanding immediate recovery (like in the simpler robustness game ROB discussed in Section A), we allow a factor of β gap. Concretely, if entropy γ^* took T^* steps to accumulate, we demand recovery by time $T \leq \beta T^*$.

▶ **Definition 3.** *The advantage of a* (q, τ)*-attacker* $\mathcal{A}$ *in the* $\mathsf{NROB}(\gamma^*, \beta, q)$ *game is denoted by* $\mathrm{Adv}^{\mathrm{NROB}}_{\mathsf{PRNG}}(\mathcal{A})$*. Further, we say that* PRNG *is* $(\gamma^*, \beta, q, \epsilon, \tau)$*-secure if for any* (q, τ)*-attacker* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\mathrm{NROB}}_{\mathsf{PRNG}}(\mathcal{A}) \leq \epsilon.$$

Game The PRNG Robustness* Game

```
NROB
  $\sigma = \perp$; $s = 0$; $c = 0$
  $b \leftarrow \{0,1\}$; $\mathsf{corrupt} = \mathsf{true}$
  $T = 0; T^* = 0$
  for $i = 1, \ldots, q$ do
      $(\sigma, X_i) \leftarrow \mathcal{A}^{\mathbf{get-next},\mathbf{get-state},\mathbf{next-ror}}(\sigma)$
      $x_i \leftarrow X_i$
      $s = \mathsf{refresh}(s, x_i)$
      $c = c + \mathrm{H}_\infty(X_i)$
      $T = T + 1$
      if $c \geq \gamma^*$ then
          if $T^* = 0$ then
              $T^* = T$
          if $T \geq \beta T^*$ then
              $\mathsf{corrupt} = \mathsf{false}$
  $b' \leftarrow \mathcal{A}(\sigma)$

get − next
  $(s, y) = \mathsf{next}(s)$
  return $y$

next − ror
  $(s, y_0) = \mathsf{next}(s)$
  $y_1 \leftarrow \{0,1\}^r$
  if $\mathsf{corrupt} = \mathsf{true}$ then
      return $y_0$
  return $y_b$

get − state
  $c = 0$ ; $\mathsf{corrupt} = \mathsf{true}$
  $T = 0; T^* = 0$
  return $s$
```

Figure 1 The Robustness Game with Premature Next Calls $\mathsf{NROB}(\gamma^*, \beta, q)$. This is in contrast to the Robustness Game without Premature Next Calls which is presented in Figure 5.

We refer the readers to the works of Dodis et al. [8] and Coretti et al. [4] for discussions on different security models. For comparison, though, we provide the formal definition of the simpler ROB notion in Section A. The critical difference between ROB and NROB is that the former resets the entropy counter $c = 0$, if an adversary invokes **get − next** when $\mathsf{corrupt} = \mathsf{true}$. Additionally, ROB implicitly sets $\beta = 1$, meaning immediate recovery when enough entropy enters the system after the compromise (or any premature next call).

3.2 Impossibility Result

The idea of our attack is that the adversary provides bit inputs such that every n inputs has one bit of entropy. Further, the premature next call will reveal information about this bit. We will prove the result through a series of lemmas. As mentioned before, we will assume that the inputs and the outputs are merely bits.

In the remainder of this section, we will work with a function $f_{\mathsf{PRNG}} : \{0,1\}^\mu \times \{0,1\}^n \to \{0,1\}$, for $\mathsf{PRNG} = (\mathsf{refresh}, \mathsf{next})$. This function $f_{\mathsf{PRNG}}(s, x)$ represents the application of n iterated refresh calls, starting from an initial state s with input $x_1, \ldots, x_n \in \{0,1\}$, before finally applying next to produce an output bit y, or more formally:

```
$f_{\mathsf{PRNG}}(s, x_1 || \ldots || x_n)$:
    for $i = 1$ to $n$
        $s = \mathsf{refresh}(s, x_i)$
    $(s, y) = \mathsf{next}(s)$
    return $y$
```

This is equivalent to applying one "big-refresh" before one next, as indicated before. Further, we write x_{-i} for $x_1 || \ldots || x_{i-1} || x_{i+1} || \ldots || x_n$, i.e., the binary string x, except for the i-th bit. Then, we can define $x_{-i,\chi}$ to be the string where the i-th bit is set to χ, i.e. $x_{-i,\chi} := x_1, \ldots, x_{i-1}, \chi, x_{i+1}, \ldots, x_n$. For any function g and any i, we abuse notation and write $g(x_{-i,\chi})$ as a shorthand for $g(x_1 || \ldots || x_n)$ where i-th bit is χ. We will also use X to denote the random variable corresponding to $x_1 || \ldots || x_n$ and use X_{-i} to denote the random variable corresponding to x_{-i}.

▶ **Lemma 4.** *There exists a randomized $O(n^2)$ algorithm* $\textsc{Find}^g$ *with oracle access to any function* $g : \{0,1\}^n \to \{0,1\}$*, such that with probability at least* $1 - 2^{-n}$ *(over the coins* $\textsc{Find}^g$*),* $\textsc{Find}^g$ *outputs* (i, z) *which satisfies precisely one of the following two (disjoint) properties:*

- $i = 0$*,* $z \in \{0,1\}$*, and* $\mathsf{P}[g(U_n) = z] \geq 0.6$*.*
- $1 \leq i \leq n$*,* $z \in \{0,1\}^{n-1}$ *and* $g(x_{-i,0}) \neq g(x_{-i,1})$ *where* $x_{-i} = z$*.*

(In other words, $\textsc{Find}^g$ *either discovers that* $g(U_n)$ *is biased, or it identifies two* n*-bit strings that differ in a single bit such that* g *returns different values on these two strings.)*

Proof. The algorithm $\textsc{Find}^g$ is defined in Figure 2. The $\textsc{Find}^g$'s output satisfies case 2 unless , after n^2 tries, the algorithm fails to find a value in the first loop. Further, in the second loop, the algorithm merely outputs the majority element.

Analysis of First for Loop. Let us look at trying to determine i, z such that it satisfies the second property. To this end, we will rely on results from graph theory. Specifically, we will use the edge isoperimetric inequality for a Hypercube graph [13, §4], which we recall (in our context) below.

For our setting, we have a Hypercube graph $Q_n = (V, E)$ where each vertex corresponds to a binary vector of length n, i.e., $|V| = 2^n$. Further E is the set of all edges that connects $(\mathbf{u}, \mathbf{v})$ if the Hamming distance between $\mathbf{u}$ and $\mathbf{v}$ is exactly 1. This gives us that: $|E| = n \cdot 2^{n-1}$. Now, we are interested in edges between a vertex $\mathbf{u}$ and $\mathbf{v}$ if $g(\mathbf{u}) \neq g(\mathbf{v})$. Now, for any set S of size $k \leq 2^{n-1}$, the number of "cut" edges C from the set to its compliment is bounded by the isoperimetric inequality [13, §4.2.1] as follows:

$$C \geq k \cdot (n - \log_2 k) \geq k$$

However, now we need to determine how many $\mathbf{u} \in V$ exists such that $g(\mathbf{u}) = 0$ (or 1). If, $0.4 \leq \mathbb{E}[g(U_n)] \leq 0.6$, then we know that there exists $0.4 \cdot 2^n$ vectors $\mathbf{u}$ with $g(\mathbf{u}) = 0$ and a similar number for $g(\mathbf{u}) = 1$.

Therefore, the probability of choosing the desired edge is at least:

$$\frac{k}{n \cdot 2^{n-1}} \geq \frac{0.4 \cdot 2^n}{n \cdot 2^{n-1}} = \frac{0.8}{n}$$

In other words, the probability that a randomly chosen edge is the desired edge occurs with probability $p \geq 0.8/n$. Therefore, one can simply pick an edge $e \in E$, uniformly at random, and then test to see if it is the desired edge. Now, if one were to do n^2 such tests, we get:

$$\mathsf{P}[g(x_{-i,0}) \neq g(x_{-i,1})] > 1 - 2^{-n}$$

This math follows from the fact that the probability of failure of algorithm is:

$$\left(1 - \frac{0.8}{n}\right)^{n^2} \leq e^{-0.8n} < 2^{-n}$$

Note that this result only follows if $0.4 \leq \mathbb{E}[g(U_n)] \leq 0.6$.

Analysis of Second for Loop. However, if $\mathbb{E}[g(U_n)] < 0.4$ or $\mathbb{E}[g(U_n)] > 0.6$, then we know that the distribution, is biased either in favor of 0 or 1. If it is biased in favor of 1 (i.e., $\mathbb{E}[g(U_n)] > 0.6$), then we know that $> 0.6 \cdot 2^n$ inputs $\mathbf{x}$ will evaluated to 1 or $< 0.4 \cdot 2^n$. In other words, the probability of success $p > 0.6$. Therefore, one can apply Chernoff bounds, to get that $\mathsf{P}[g(U_n) = z] \geq 0.6$ with probability $1 - 2^{-n}$.

The correctness of $\textsc{Find}^g$ follows from our earlier discussion. It is easy to see that $\textsc{Find}^g$ runs in time $O(n^2)$ as the lines inside the first for loop take constant time if one were to sample the edge by picking i and x_{-i}. ◀

```
Algorithm FIND$^g$
for $i = 1$ to $n^2$:
    Pick an edge $(\mathbf{u}, \mathbf{v}) \in E$, uniformly at random.
    Use oracle access to $g$ to compute $g(\mathbf{u})$ and $g(\mathbf{v})$.
    if $g(\mathbf{u}) \neq g(\mathbf{v})$ then
        Find $i$ such that $u_i \neq v_i$.
        By definition, there exists a unique $i$ that satisfies this condition.
        return $(i, u_{-i})$
        break
for $i = 1$ to $120 \cdot n$:
    $count = 0$
    Sample $\mathbf{x} \leftarrow \{0,1\}^n$
    Compute $count = count + g(\mathbf{x})$
if $count > n/2$ then $z = 1$
else $z = 0$
return $(0, z)$
```

Figure 2 Description of FIND^g.

▶ **Theorem 5.** *There is no* $(\gamma^*, \beta, q, 0.1, \tau)$*-secure PRNG for* $\gamma^*\beta < \sqrt{q}$ *and* $\tau \geq \Omega((t_{\mathsf{next}} + t_{\mathsf{refresh}}) \cdot n^3)$ *where* $n = \gamma^* \cdot \beta$ *and* t_{next} *and* t_{refresh} *are the time required to compute* next *and* refresh *respectively.*

Proof. We will use the FIND^g algorithm defined in Lemma 4 to create an adversary $\mathcal{A}$ that wins the NROB(γ^*, β) security game. The pseudocode for the adversary is provided in Figure 3. Here, the definition of the function g is as follows: $g(\mathbf{x}) = f(s, x_1||\ldots||x_n)$ where s is the current state s and $n = \gamma^* \cdot \beta$. $\mathcal{A}$ is aware of the very first state s. The attacker then runs FIND^g on this function g and receives (i, z) as output. Now, we have two cases:

- $i = 0$. Recall that $i = 0$ implies that $g(U_n)$ is biased towards the value z. Therefore, $\mathcal{A}$ simply invokes **get − state** first. This is done not to retrieve the state, but rather to reset the counters of T and T^*. Now, $\mathcal{A}$ uses the biased nature of g on U_n to provide uniform bit $n = \gamma^*\beta$ times. At the end of this process, we have $T^* = \gamma^*\beta$ and the attacker is required to break the scheme within another β steps. After the n inputs, $\mathcal{A}$ invokes **next − ror** to receive its challenge response. If this challenge response is equal to z, then we know that $b = 0$, indicating it is the real distribution and not the random distribution.
- $i \neq 0$. Recall that $i \neq 0$ implies that there exists two n-bit strings that differ in one bit, but g produces different evaluations. i is the bit where the strings differ and z is the value for the remaining bits. $\mathcal{A}$ begins by writing down z in its state, and then provides one bit of entropy by randomly sampling x_i. Now, $\mathcal{A}$ uses a "premature" call to **get-next** and receives y as response. With knowledge of z, $\mathcal{A}$ can compute g for two choices of input at the i-th bit and then use y to uniquely determine what was the input at x_i which also helps $\mathcal{A}$ recover the state. This process is repeated γ^* times to provide γ^* bits of entropy. We keep doing this for ${\gamma^*}^2\beta^2$ steps, and then, request **next-ror** . However, with knowledge of the state, due to premature next, $\mathcal{A}$ knows the challenge and therefore wins with a non-negligible advantage.

In other words, we have an attacker which can break this scheme, with non-negligible probability, if $q > {\gamma^*}^2\beta^2$.[2] ◀

[2] Note, that when $q < \gamma^*\beta$, every PRNG is vacuously secure as there is no need for recovery: at least γ^* steps are needed to inject the required γ^* bits of entropy, and the attacker simply runs out of refresh calls to trigger the security requirement. This, of course, assumes ideal entropy accumulation.

```
Algorithm $\mathcal{A}$

Set $s = 0$
$\sigma = \perp$
$t = 0$
while $t \leq \gamma^{*2}\beta^2$
    Set $g(\mathbf{x}) = f(s, \mathbf{x})$
    $(i, z) \leftarrow \textsc{Find}^g$
    if $i = 0$ then
        Invoke get-state to get the current state $s^*$.   // This resets $T = 0$.
        for $j = 1$ to $n$:
            Output $X_{t+j} = U_1$                          // $\mathrm{H}_\infty(X_{t+j}) = 1$.
        Invoke next-ror for challenge $\delta$
        if $\delta = z$ then return $0$
        else return $1$
    else
        Set $X_{t+i} = U_1$                    // $\mathrm{H}_\infty(X_{t+i}) = 1$.
        Use $z$ to set $X_{t+k}$ for $k \neq i$.    // $\mathrm{H}_\infty(X_{t+k}) = 0$ for $k \neq i$.
        Invoke get-next to get output $y$.
        Let $a_{-i} = z$
        if $g(a_{-i,0}) = y$ then $x_{t+i} = 0$
        else $x_{t+i} = 1$
        for $i = 1$ to $\alpha\beta$
            $s = \mathsf{refresh}(s, x_{t+i})$
        $(s, y) = \mathsf{next}(s)$
        $t = t + \alpha\beta$
Invoke next-ror for challenge $\delta$
if $\mathsf{next}(s) = (\cdot, \delta)$ then return $0$
else return $1$
```

Figure 3 Pseudocode for $\mathcal{A}$ for Theorem 5.

3.3 Towards Positive Results

The impossibility is, of course, artificial, but it raises questions about how to overcome it, even assuming ideal entropy accumulation and extraction. In Section 4 we abstract the notion of the scheduler which models security against premature next attacks using multiple pools which assume to accumulate entropy optimally (which abstracts away entropy accumulation and extraction).

In this setting, we will first analyze a single-pool scheduler scheme for the special "root pool" in Section 5. This scheme uses a single pool with exponentially decaying time intervals to drain this pool, but the rate of such recovery will depend on the entropy rate *counted from the boot time* (as opposed to the latest compromise in the general notion). The latter point is why we don't want to use this one-pool scheme for the general-purpose PRNG, where we would like to recover from compromise *no matter when it happens*.

For such scenarios, we revisit the round-robin Fortuna scheduler, where [8] observe that this scheme provably overcomes our impossibility result, by assuming all entropy comes at a fixed (but unknown) rate. Instead, in Section 6 we significantly generalize this positive result. The idea is to redefine the notion of entropy we use in a way that makes it more restrictive than traditional (min-) entropy, but not as restrictive as assuming fixed constant rate.[3] Intuitively, our notion of entropy will not allow attacks where the entropy varies too widely within a given round-robin (but can change from one round-robin to another) – in a sense that the attacker will get almost no credit for high-entropy samples when there is at least one low entropy sample within a given round-robin.

[3] We also note that the results about fast entropy accumulation in the register [5] might justify why our new (more restrictive) notion of entropy might be reasonable to expect in practice.

Construction: Premature-Next Robust PRNG

```
refresh*(x, s̄)
    Parse $\overline{s}$ as $(s_0, \ldots, s_{k-1}, \rho)$
    $\mathsf{in}, \mathsf{out} \leftarrow \mathsf{SC}()$
    $s_{\mathsf{in}} \leftarrow \mathsf{refresh}(s_{\mathsf{in}}, x)$
    if $\mathsf{out} \neq \perp$ then
        $(s_{\mathsf{out}}, R) \leftarrow \mathsf{next}(s_{\mathsf{out}})$
        $\rho \leftarrow \rho \oplus R$
    return $\overline{s} = (s_0, \ldots, s_{k-1}, \rho)$

next*(s̄)
    Parse $\overline{s}$ as $(s_0, \ldots, s_{k-1}, \rho)$
    $(Y, \rho) \leftarrow \mathcal{G}(\rho)$
    return $\big(\overline{s} = (s_0, \ldots, s_{k-1}, \rho), Y\big)$
```

Figure 4 Construction of $\mathcal{G} = (\mathsf{refresh}^*, \mathsf{next}^*)$.

4 Seedless Scheduler

For the remainder of this paper, we will assume ideal accumulation and extraction. Further, rather than working with entropy, we will employ the notion of a sequence of weights $\mathbf{w} = (w_1, \ldots, w_q)$ where the weights have been normalized so that $w_i \in [0, 1]$ and a pool is "full" when it has accumulated weight 1. (Specifically, to move between the weight w_i and the entropy γ, one should multiply by the entropy ${\gamma^*}_{\mathsf{rob}}$ required for a single pool to recover.) See [8].

4.1 Syntax of a Scheduler

We define the syntax of the scheduler below. Note that this scheduler is deterministic and oblivious, i.e., it does not depend on the actual input or its entropy.

▶ **Definition 6** (Syntax of Scheduler). *A (k, q)-scheduler is a deterministic algorithm* SC *that produces q pairs: $\{(\mathsf{in}_i, \mathsf{out}_i)\}_{i=1}^q$ where $\mathsf{in}_i \in [k], \mathsf{out}_i \in [k] \cup \{\perp\}$ for $i = 1, \ldots, q$.*

Note that, when the number of "pools" k is not critical to be specified explicitly, a deterministic (k, q)-scheduling scheduler can be thought of as a sequence of values $\{\mathsf{empty}\}_{i=1}^q$ corresponding to the time at which each input i with weight w_i is emptied. More formally, we can define: $\mathsf{empty}_i := \min \{j : j > i \wedge \mathsf{out}_j = \mathsf{in}_i\}$

4.2 Seedless PRNG, with Premature Next

Before we venture into the security of such a scheduler, it would be prudent to take a step back and look at an informal composition of a seedless scheduler with PRNGs that are not resilient to premature next in order to achieve security with premature next. Indeed, it is also equally important to frame our composition results, in the face of the impossibility result from Section 3.2 (and also the unrestricted scheduler impossibility later in this section). This is precisely the reason why we do not state a formal composition theorem, as it is vacuous for the most general case. However, the composition is still robust for restricted notions of scheduler security to yield relaxed forms of PRNG security with premature next.

The composition relies on seedless PRNGs which are not secure with premature next. These are typically parametrized by just γ^*, which is the minimum entropy needed for the PRNG to begin producing pseudorandom outputs (see Figure 5). In essence, these have $\alpha = \gamma^*$ and $\beta = 1$ with a reset of all counters when an adversary invokes **get** − **next** with $\mathsf{corrupt} = \mathsf{true}$. The instantiation of this PRNG can be from the work of Coretti et al. [4]

or from the work of Dodis et al. [6]. Such a PRNG, secure without premature next and parametrized by γ^* is combined with a scheduler. The goal of a scheduler would be to ensure that the input, as it arrives, is allocated a particular pool such that:

- With "enough entropy", a pool is filled, i.e., accumulates γ^* amount of entropy.
- This pool will be emptied within "sufficient time", to recover from compromise.

We will formalize these notions of "enough entropy" and "sufficient time" in the next section.

Formally, we define a seedless PRNG, with construction as follows:

- Let SC be a scheduler with k pools.
- Let $\mathcal{G}_i = (\mathsf{refresh}_i, \mathsf{next}_i)$ be *seedless* PRNGs with input (see Section A), for $i = 0, \dots, k-1$. For simplicity, we will assume that each $\mathcal{G}_i$ is (m, r)-PRNG. These are PRNGs which are not secure with premature next calls.
- Let $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{2m}$ be a pseudorandom generator (without input).

Then, we construct a PRNG with input $\mathcal{G}(\mathsf{SC}, \{\mathcal{G}_i\}_{i=0}^{k-1}, \mathbf{G}) = (\mathsf{refresh}^*, \mathsf{next}^*)$ as shown in Figure 4, where the scheduler mandates which pool $\mathcal{G}_{\mathsf{in}}$ to use (via $\mathsf{refresh}$) to accumulate entropy from a new sample, and which pool $\mathcal{G}_{\mathsf{out}}$ (if any) to "empty" (via next) into the main register ρ for $\mathbf{G}$.

4.3 Security of a Scheduler

We will define different notions of security for a scheduler. As with PRNGs, (k, q)-scheduler security model is parameterized by two parameters α, β. Informally, it states that if the adversary chooses to provide α units of fresh entropy (i.e., a sequence of w_i values that sum up to α) within a time $t \le q/\beta$, then we guarantee recovery within time $\beta \cdot t \le q$. Formally,

▶ **Definition 7** (General Security of Scheduler). *A (k, q)-scheduler is (α, β)-general-secure if if $\forall t_0, t$ such that $t_1 = t_0 + \beta \cdot t \le q$, and $\forall$ weights $w_1, \dots, w_q \in [0, 1]$ such that $\sum_{i=1}^{t} w_{t_0+i} \ge \alpha$, the scheme recovers from the compromise in time $t_0 + \beta t$ where recovery occurs if $\exists\ j \in [k]$ and $\exists\ \widehat{T} \in [t_0 + 1, t_0 + \beta \cdot t]$ such that:*

1. $\mathsf{out}_{t_0+1}, \dots, \mathsf{out}_{\widehat{T}-1} \ne j$ *(pool j has not been emptied before time $\widehat{T}$);*
2. $\mathsf{out}_{\widehat{T}} = j$ *(pool j is emptied at time $\widehat{T}$); and*
3. *(pool j has filled)* $\sum_{\substack{t_0 < i \le \widehat{T} \\ \mathsf{in}_i = j}} w_i \ge 1$.

4.4 Impossibility Result

We can show that, for general security, there exists an impossibility result. Specifically, we will show that for any $k \in \mathbb{N}$, there exists a choice of q such that any (k, q)-scheduler is not (α, β)-secure. In other words, for a suitable choice of q, one can break the scheduler to never recover from compromise for any α, β. Note that this is incomparable to the earlier impossibility result discussed in Section 3.2 as this assumes the existence of pools.

▶ **Theorem 8.** *For any $k \in \mathbb{N}$, there exists $q^* = \alpha^2\beta^2$ such that a given (k, q)-scheduler is not (α, β)-secure for any $q \ge q^*$.*

We defer the proof of this theorem to Section B. The immediate consequence of this impossibility result is the following: there exists input weight sequence $\mathbf{w}$ such that, irrespective of the number of pools, we can inject entropy at a slow rate, such that no scheduler is general-secure. It also implies that we need to make some relaxations to achieve usable security results.

5 Reboot Secure Schedulers

The first relaxation corresponds to the situation when the system is just rebooted, i.e., we are at $t_0 = 0$. We will call this as the "reboot security" of a scheduler. This corresponds to the situation when you just turn on the computer. For this case, we can have a much simpler and better RNG, having only one pool. Like Fortuna, this pool is emptied every β^i steps for gradually increasing values of $i = 0, 1, 2, \ldots$, where β is a small integer (Windows 10 uses $\beta = 3$).

▶ **Definition 9** (Reboot Security of Scheduler). *A (k, q)-scheduler is (α, β)-reboot-secure if for $t_0 = 0$, $\forall t$ such that $t_1 = t_0 + \beta \cdot t \leq q$, and $\forall$ weights $w_1, \ldots, w_q \in [0, 1]$ such that $\sum_{i=1}^{t} w_{t_0+i} \geq \alpha$ the scheme recovers from the compromise in time $t_0 + \beta t$, where the definition of recovery is as defined in Definition 7.*

The composition of such a reboot-secure scheduler with our "not-premature-next" PRNGs will trivially yield a "premature-next" boot PRNG, i.e., the PRNG that is used at the time when the system is booting up.

We start with a lower bound on reboot-security, irrespective of the number of pools k.

▶ **Theorem 10.** *For a (k, q)-scheduler to be (α, β)-reboot secure, $\alpha \geq \lfloor \log_\beta(q) - \log\log q \rfloor - 1$ (i.e., $q \leq \alpha\beta^\alpha$)*

For simplicity let us assume that $q = \alpha\beta^{\ell+1}$, for some $\ell > 0$. Then, divide the time from $\alpha + 1$ to q into intervals of the following form: $(\alpha\beta^{i-1}, \alpha\beta^i]$ for $i = 1$ to $\ell + 1$. We have the following claim:

▷ Claim 11. For any (α, β)-reboot secure scheduler with corresponding emptying sequence $\mathsf{empty}_1, \ldots, \mathsf{empty}_q$ and any $i \in [\ell]$, there must exist a t such that $\mathsf{empty}_t \in (\alpha\beta^i, \alpha\beta^{i+1}]$. (In other words, there must be a pool that is first emptied after roughly β^i steps for every i.)

We defer the proof of this claim to Section B.

Proof of Theorem 10. From Claim 11, we get that there are at least $\lfloor \log_\beta(q/\alpha) \rfloor$ distinct empties, and there needs to be entropy of 1 emptied in each of these empties. By Pigeonhole Principle, we will need $\alpha \geq \lfloor \log_\beta(q/\alpha) \rfloor$ to have any hope of recovery, which implies $\alpha \geq \lfloor \log_\beta(q) - \log\log q \rfloor - 1$. ◀

We now give a scheme that nearly matches the lower bound. This scheme uses the same strategy as Windows 10's "Root RNG" which is used at system startup [9].

▶ **Construction 12** (Reboot Scheme). *The scheme has $k = 1$. $\mathsf{in}_i = 0$ for $i = 1, \ldots, q$.*

$$\mathsf{out}_i = \begin{cases} 0 & \textbf{if } i = \beta^j \\ \perp & \textbf{else} \end{cases}$$

In other words, $\forall i \in [\beta^{j-1}, \beta^j)$, empty at time β^j.

▶ **Theorem 13.** *Construction 12 is (α, β)-reboot secure for $q = \alpha\beta^\alpha$ (i.e., $\alpha \approx \log_\beta q - \log\log q$).*

Proof. Define t to be the time within which the adversary provides α entropy, i.e., $\sum_{i=1}^{t} w_i \geq \alpha$ where these w_i are adversarially chosen. It is clear that $t \geq \alpha$, as we need at least α steps to provide α entropy when $w_i \in [0, 1]$.

Let i be such that $\alpha \in (\beta^{i-1}, \beta^i]$. Now, it is clear that if $t = \alpha$, then the empty at β^i will ensure recovery from compromise. We can induct similar to the proof of Claim 11 to get that if $t \in [\beta^{\ell-1}, \beta^\ell)$ for some $\ell \geq i$, then there $\exists j \in [\beta^{\ell-1}, \beta^\ell)$ such that $w_j = 1$ (or possibly a set of such j's which sum up to 1), which is emptied at β^i, thus recovering from compromise. Specifically, if we have $t \in [\beta^{\ell-1}, \beta^\ell)$, then at each of the preceding $\ell - 1$ intervals (each with an empty), $\mathcal{A}$ provides $1 - \epsilon$ entropy, for some arbitrarily small ϵ. This gives a total of almost $\ell - 1$ entropy across these intervals. Therefore, it follows that the remainder of $\alpha - \ell + 1 > 1$ needs to be provided between $w_{\beta^{\ell-1}}$ and w_t to hit α and all of these are emptied at β^ℓ, recovering from compromise. ◀

6 Repeat Secure Schedulers

A general secure scheme is a stronger model of security than the reboot model. This follows because the value of t_0 is also the choice of the adversary, in addition to the choice of t. However, the impossibility result from Theorem 8 imply a need for relaxation.

Round-Robin Schedulers. Simple round-robin schedulers achieve very good $\alpha \approx \log_\beta(q)$ for the special cases when all of the w_t are equal to some (unknown, adversarially chosen) value w, i.e., $w_1 = w_2 = \ldots = w_q = w$ and setting the number of pools $k \approx \log_\beta(q)$ (so 1 or 2 pools are too little). β is a smaller integer usually 2 or 3 in practice, as in [10, 8]. More formally, such schedulers simply set $\mathsf{in}_t = t \bmod k$ As for out_t, this is set to $\perp$ inside one round (i.e. $t \bmod k \neq 0$). At the the of each round, when $t = k\ell$, one looks at the largest index $i \geq 0$ such that β^i divides ℓ. Then out empties the i-th pool: $\mathsf{out}_t = i$

▶ Remark 14. There is a marginal gain in efficiency when we empty all pools $\leq i$, instead of just the i-th pool. In other words, out is a set, rather than a single index. However, for our analysis below, we will continue to work with the assumption that a single pool is emptied. (More generally, we do not make much of an attempt to optimize the parameters that we achieve. See [8] for an optimized version of similar construction.)

k-smooth Sequences. Our main observation is that we can significantly extend the constant-rate analysis as follows. The idea is to allow support any constant rate within a round-robin (rather than go for a constant (but unknown) rate scheduler). This constant can change arbitrarily once the next round-robin is started. Namely, we don't have to fix the same constant for all q entropies but can change it every $k \ll q$ steps. In practice, this means that while the quality of entropy can change over time, we heuristically assume that it changes rather smoothly, and we rarely have huge jumps within a given round-robin.

▶ **Definition 15** (Repeating Sequences). $\mathbf{w} = (w_1, \ldots, w_q)$ *with* $0 \leq w_i \leq 1$ *is called* k-*repeating if* $w_{jk+1} = w_{jk+2} = \ldots = w_{jk+k}$ *for* $j = 0, \ldots, t-1$ *where* $q = k \cdot t$

▶ **Definition 16** (Repeat Security of Scheduler). *A* (k, q)-*scheduler is* (α, β, k)-*repeat-secure if* $\forall$ t_0, t *such that* $t_1 = t_0 + \beta \cdot t \leq q$, *and* $\forall$ k-*repeating weights* $w_1, \ldots, w_q \in [0, 1]$ *such that* $\sum_{i=1}^{t} w_{t_0+i} \geq \alpha$ *the scheme recovers from the compromise in time* $t_0 + \beta t$, *where the definition of recovery is as defined in Definition 7.*

To achieve such repeating sequences, we take any standard $\mathbf{w} = (w_1, \ldots, w_q)$ and apply a k-flattening, as defined below.

▶ **Definition 17** (k-Flattening). *Given a sequence* $\mathbf{w} = (w_1, \ldots, w_q)$ *and a number* $k \geq 1$, *where for simplicity of notation let us assume* $q = kt$, *we define* k-smooth flattening *of* $\mathbf{w}$ *to be* $\mathbf{w}' = (w'_1, \ldots, w'_q)$, *where for any round-robin* $j \in \{0, \ldots t-1\}$ *and* $i \in \{1 \ldots k\}$, *we let*

$$w'_{jk+i} = \min(\ w_{jk+1},\ w_{jk+2}, \ldots,\ w_{(j+1)k}\ )$$

Intuitively, we change the entropy w_j to the smallest of k surrounding entropies inside a given round-robin. Of course, $k = 1$ corresponds to $w'_t = w_t$, but we already know that 1 pool is not enough (as this would give a general scheduler for the unrestricted entropy setting). For larger k, however, the flattened values could be noticeably lower than the original. For example, if $k = 3$ and $\mathbf{w} = \{1, 1/2, 1/3, 1/4, 1/5, 1/6\}$, the 3-flattening of $\mathbf{w}$ is $\mathbf{w}' = \{1/3, 1/3, 1/3, 1/6, 1/6, 1/6\}$. Of course, for a constant rate $w_1 = \ldots w_q = w$, k-flattening does not change anything, which explains why our results below naturally generalize the constant-rate analysis from the work of Dodis et al. [8].

Jumping ahead, we will see that the Fortuna scheduler is "secure" for any (normalized) entropy sequence $\mathbf{w}$, with the understanding that the attacker gets "entropy credit" within a single round-robin equals to k times the *lowest* entropy value in contributes within this round.

New Result. Now, we show that while the original (α, β)-definition above cannot be achieved when applied to $\mathbf{w}$ itself, the analysis for constant-rate schedulers works for general entropy sequences, provided we simply apply it to k-flattening of $\mathbf{w}$ (where $k \approx \log_\beta q$ is the number of pools) instead of $\mathbf{w}$ itself! Namely, a given round only gets "credit" for the smallest entropy (times k) it contributed to any of the k pools. So we do not give the adversary credit if it wildly changes the entropy values within a given round.

We now present our construction, which is parameterized by the number of pools k and a base b. One typically takes $b = 2$ or $b = 3$, and, e.g., $k = 32$ or $k = 64$ in practice, and works for $q \leq b^k$.

▶ **Construction 18** (Smooth scheduler). *Consider the following* $(k, q := b^k)$*-scheduler for integers* $b \geq 2$ *and* $k \geq 1$:

- $\mathsf{in}_i = i \mod k$
- $\mathsf{out}_i = \begin{cases} \perp & \textbf{if } i \mod k \neq 0 \\ j & \textbf{if } i = k\ell \end{cases}$ *where* $j \geq 0$ *is the largest* j *such that* $\ell \mod b^j = 0$ *for* $i = k\ell$

We now prove that this scheduler is secure (against k-repeating sequences). For simplicity, we make little attempt to optimize the parameters. See [8] for a carefully optimized version of this result for the special case where the entropy rate is constant (i.e., the case of q-repeating weights).

▶ **Theorem 19.** *For any integers* $b \geq 2$ *and* $k \geq 1$, *Construction 18 is* (α, β, k)*-repeat-secure for*

$$\alpha := 3k - 2 \approx 3\log_b q; \qquad \textit{and} \qquad \beta := 2b\left(1 + \frac{k}{\alpha}\right) \approx \frac{8b}{3} = \frac{8}{3} \cdot q^{1/k}$$

In particular, for $k = \log_b q$ *and* $q \geq b^2$, *we have* $\alpha \leq 3\log_b q$ *and* $\beta \leq 3b$.

Notice, this result explains how the recovery factor β shrinks very quickly as we increase the number of pools k, starting with (roughly) q all the way down to being a constant. In particular, β becomes constant once the number of pools becomes logarithmic in q.

Moreover, up to constant factors in α and β (which, again, we do not attempt to optimize), Theorem 19 is tight. In particular, [8, Proposition 1] proved that even in the "constant-rate" case of q-repeating weights, no scheduler can be (α, β)-secure with $\alpha\beta \le \log_e q - \log_e \log_e q - 1$. And our scheduler matches this bound (up to a constant factor) when $b = O(1)$ and $k = O(\log q)$. We defer the proof of the theorem to Section B.

References

1 Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 2005*, pages 203–212, Alexandria, Virginia, USA, November 7–11 2005. ACM Press. doi:10.1145/1102120.1102148.

2 Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A, 2012.

3 Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM J. Comput.*, 17(2):230–261, 1988. doi:10.1137/0217015.

4 Sandro Coretti, Yevgeniy Dodis, Harish Karthikeyan, and Stefano Tessaro. Seedless Fruit is the sweetest: Random number generation, revisited. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 205–234, Santa Barbara, CA, USA, August 18–22 2019. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-26948-7_8.

5 Yevgeniy Dodis, Siyao Guo, Noah Stephens-Davidowitz, and Zhiye Xie. No time to hash: On super-efficient entropy accumulation. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 548–576, Virtual Event, August 16–20 2021. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-84259-8_19.

6 Yevgeniy Dodis, Siyao Guo, Noah Stephens-Davidowitz, and Zhiye Xie. Online linear extractors for independent sources. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic Cryptography, ITC 2021, July 23-26, 2021, Virtual Conference*, volume 199 of *LIPIcs*, pages 14:1–14:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITC.2021.14.

7 Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 647–658, Berlin, Germany, November 4–8 2013. ACM Press. doi:10.1145/2508859.2516653.

8 Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too: Optimal recovery strategies for compromised rngs. *Algorithmica*, 79(4):1196–1232, 2017. doi:10.1007/s00453-016-0239-3.

9 Niels Ferguson. The windows 10 random number generation infrastructure, October 2019. URL: https://aka.ms/win10rng.

10 Niels Ferguson and Bruce Schneier. *Practical cryptography*. Wiley, 2003.

11 Peter Gazi and Stefano Tessaro. Provably robust sponge-based PRNGs and KDFs. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 87–116, Vienna, Austria, May 8–12 2016. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-49890-3_4.

12 Viet Tung Hoang and Yaobin Shen. Security analysis of NIST CTR-DRBG. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 218–247, Santa Barbara, CA, USA, August 17–21 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-56784-2_8.

13 Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *BULL. AMER. MATH. SOC.*, 43(4):439–561, 2006.

14 Daniel Hutchinson. A robust and sponge-like PRNG with improved efficiency. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 381–398, St. John's, NL, Canada, August 10–12 2016. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-69453-5_21.

15 John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *In Sixth Annual Workshop on Selected Areas in Cryptography*, pages 13–33. Springer, 1999.

16 John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In Serge Vaudenay, editor, *FSE'98*, volume 1372 of *LNCS*, pages 168–188, Paris, France, March 23–25 1998. Springer, Heidelberg, Germany. doi:10.1007/3-540-69710-1_12.

17 Noam Nisan and David Zuckerman. Randomness is linear in space. *J. Comput. Syst. Sci.*, 52(1):43–52, 1996. doi:10.1006/jcss.1996.0004.

18 Amit Sahai and Salil Vadhan. A complete problem for statistical zero knowledge. *Journal of the ACM*, 50(2):196–249, 2003. Extended abstract in FOCS '97.

19 Thomas Shrimpton and R. Seth Terashima. A provable-security analysis of Intel's secure key RNG. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 77–100, Sofia, Bulgaria, April 26–30 2015. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-46800-5_4.

20 Wikipedia contributors. /dev/random – Wikipedia, the free encyclopedia, 2021. [Online; accessed 9-January-2022]. URL: https://en.wikipedia.org/w/index.php?title=/dev/random&oldid=1056079736.

21 Joanne Woodage and Dan Shumow. An analysis of NIST SP 800-90A. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 151–180, Darmstadt, Germany, May 19–23 2019. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-17656-3_6.

A PRNG Robustness, without Premature Next

This is an abridged discussion about the robustness security game ROB, for the seedless setting (and with independent samples), but *without* allowing Premature Next calls. The security game is presented as Figure 5. The main difference from the NROB game presented in Figure 1 is that the entropy counter c is reset to 0 with each "premature next" call to **get** – **next**, and also there is no recovery delay parameter β. As the result, there is no need to keep track of the number of steps T^* to accumulate γ^* bits of entropy.

B Deferred Proofs

Proof of Theorem 8. The attack works as follows: we will provide α entropy, in $\alpha^2\beta$ steps. The security requirement is that recovery needs to happen within time $\alpha^2\beta^2$. Recall that recovery occurs if there is a pool that is emptied within $\alpha^2\beta^2$ which has total entropy of 1.

More formally, let I_j denote the j-th interval, of length $\alpha\beta$ starting from 0. This leads us to two cases:

- $\exists j^*$ such that no pool is emptied within I_{j^*}. Formally, there exists no time step i with $\mathsf{empty}_i \in I_{j^*}$.

  Then, set $t_0 = \alpha\beta(j^* - 1) - 1$. After this state compromise, we provide a sequence of 1s of length α, which will set $T^* = \alpha$ and expect recovery in time $t_0 + \beta T^* = \alpha\beta j^* - 1$, which is still inside the interval I_{j^*}. However, we assumed no pool is emptied within I_{j^*}, so no recovery can be possible.

Game The PRNG Robustness Game

```
ROB
  $\sigma = \bot$; $s = 0$; $c = 0$
  $b \leftarrow \{0,1\}$; corrupt = true
  for $i = 1, \ldots, q$ do
      $(\sigma, X_i) \leftarrow \mathcal{A}^{\mathbf{get-next},\mathbf{get-state},\mathbf{next-ror}}(\sigma)$
      $x_i \leftarrow X_i$
      $s = \mathsf{refresh}(s, x_i)$
      $c = c + \mathrm{H}_\infty(X_i)$
      if $c \geq \gamma^*$ then
          corrupt = false
  $b' \leftarrow \mathcal{A}(\sigma)$

get − next
  if corrupt = true then
      $c = 0$
  $(s, y) = \mathsf{next}(s)$
  return $y$

next − ror
  $(s, y_0) = \mathsf{next}(s)$
  $y_1 \leftarrow \{0,1\}^r$
  if corrupt = true then
      return $y_0$
  return $y_b$

get − state
  $c = 0$ ; corrupt = true
  return $s$
```

Figure 5 The Robustness Game (without Premature Next Calls) $\mathsf{ROB}(\gamma^*, q)$.

- $\forall j$, $\exists i$ such that $\mathsf{empty}_i \in I_j$, meaning at least one pool is emptied within all α intervals I_j.
  Set $t_0 = 0$. Then, for $j = 1, \ldots, \alpha$, pick *one* ℓ such that $\mathsf{empty}_\ell \in I_j$. Set, w_ℓ to be $1 - \epsilon$ for some arbitrarily small ϵ (remaining weights are 0). At the end of this process, the adversary has provided almost α entropy, but there is no recovery, as all of these entropies are completely wasted. By making ϵ arbitrarily small, the result follows. ◀

Proof of Claim 11. We prove this by induction. Define t to be the time within which the adversary provides α entropy, i.e., $\sum_{i=1}^{t} w_i \geq \alpha$ where these w_i are adversarially chosen. Since $w_i \leq 1$, we get that $t \geq \alpha$.

Let us assume to the contrary that there is no emptying in the interval $(\alpha, \alpha\beta]$. Now, if adversary chooses $t = \alpha$. Then, this scheme would never recover as there is no empty in the interval $(\alpha, \alpha\beta]$

Now, let us assume that there is an empty in intervals, $(\alpha, \alpha\beta], (\alpha\beta, \alpha\beta^2], \ldots, (\alpha\beta^{i-1}, \alpha\beta^i]$. We will now show that there needs to be an empty in the interval $(\alpha\beta^i, \alpha\beta^{i+1}]$. To this end, assume to the contrary. Now, note that the adversary can provide the entropy in such a way that every empty in the preceding intervals empties out $1 - \epsilon$, without recovering. This is similar to the attack detailed in the proof of Theorem 8. Further, if $t = \alpha\beta^i$, the scheme has not recovered in time 1 to t and because it has no empty in $(\alpha\beta^i, \alpha\beta^{i+1}]$ it can never hope to recover in time either. Therefore, there is an empty in the interval $(\alpha\beta^i, \alpha\beta^{i+1}]$. ◁

Proof of Theorem 19. Let $w_1, \ldots, w_q$ be k-repeating. Let t_0 and t be such that (1) $t_0 + \beta t \leq q$; and (2) $\sum_{i=1}^{t} w_{t_0+i} \geq \alpha$. We wish to show that in this case the scheduler recovers before time $t_0 + \beta t$, i.e., that there exists a $j \in [k]$ and $\widehat{T} \in [t_0 + 1, t_0 + \beta t]$ such that (1) $\mathsf{out}_{\widehat{T}} = j$; (2) $\mathsf{out}_{t_0+1}, \ldots, \mathsf{out}_{\widehat{T}-1} \neq j$; and (3) $\sum_{\substack{t_0 < i \leq \widehat{T} \\ \mathsf{in}_i = j}} w_i \geq 1$.

Indeed, we take j to be minimal such that $\mathsf{out}_{t_0+1}, \ldots, \mathsf{out}_{t_0+t} \neq j$. In particular, notice that after pool j' is emptied, pool $j' + 1$ is not emptied for the next $k(b^{j'} - 1)$ steps. And, similarly, after pool $j' + 1$ is emptied, pool j' is not emptied for the next $k(b^{j'} - 1)$ steps. It follows that $b^{j-1} \leq t/k + 1$. Since the pool j' is emptied at least once in every $2kb^{j'}$ steps, it follows that we must have $\mathsf{out}_{\widehat{T}} = j$ for some $\widehat{T} - t_0 \leq 2kb^j \leq (2t + 2k)b \leq 2b(1 + k/\alpha)t$, where in the second inequality we have used the fact that $w_i \leq 1$, which implies that $t \geq \alpha$. In particular, $\widehat{T} \leq t_0 + \beta t$, as needed.

And, since the w_i are k-repeating, we must have

$$\sum_{\substack{t_0<i\le\widehat{T}\\ \mathsf{in}_i=j}} w_i \ge \sum_{t_0<i\le\widehat{T}} \sum_{\substack{t_0<i\le t_0+t\\ \mathsf{in}_i=j}} w_i \ge \sum_{\substack{t_0'<i\le t_0'+t'\\ \mathsf{in}_i=j}} w_i = \frac{1}{k}\cdot \sum_{t_0'<i\le t_0'+t'} w_i \ ,$$

where $t_0' := \lceil t_0/k\rceil k \ge t_0$ and $t' := \lfloor t/k\rfloor k \le t$. And, since $w_i \le 1$, we trivially have that

$$\sum_{t_0'<i\le t_0'+t'} w_i \ge \sum_{t_0<i\le t_0+t} w_i - 2k + 2 \ge \alpha - 2k + 2 \ .$$

Therefore,

$$\sum_{\substack{t_0<i\le t_0+t\\ \mathsf{in}_i=j}} w_i \ge \frac{\alpha}{k} - 2 + 2/k \ge 1 \ ,$$

as needed. ◀