

# 1 On Seedless PRNGs and Premature Next

2 Sandro Coretti ✉

3 IOHK

4 Yevgeniy Dodis ✉

5 New York University, USA

6 Harish Karthikeyan ✉

7 New York University, USA

8 Noah Stephens-Davidowitz ✉

9 Cornell University, USA

10 Stefano Tessaro ✉

11 University of Washington, USA

## 12 — Abstract —

---

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

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

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

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

42 **2012 ACM Subject Classification** Security and privacy → Mathematical foundations of cryp-  
43 tography; Security and privacy → Information-theoretic techniques; Theory of computation →  
44 Pseudorandomness and derandomization

45 **Keywords and phrases** seedless PRNGs, pseudorandom number generators, PRNG, Fortuna, prema-  
46 ture next



© Sandro Coretti and Yevgeniy Dodis and Harish Karthikeyan and Noah Stephens-Davidowitz and Stefano Tessaro;

licensed under Creative Commons License CC-BY 4.0

3rd Conference on Information-Theoretic Cryptography (ITC 2022).

Editor: Dana Dachman-Soled; Article No. 9; pp. 9:1–9:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

47 **Digital Object Identifier** 10.4230/LIPIcs.ITC.2022.9

48 **Funding** *Yevgeniy Dodis*: Partially supported by gifts from VMware Labs and Google, and NSF  
49 grants 1815546 and 2055578.

50 *Stefano Tessaro*: Supported in part by NSF grants CNS-1930117 (CAREER), CNS-1926324, CNS-  
51 2026774, a Sloan Research Fellowship, and a JP Morgan Faculty Award.

## 52 **1 Introduction**

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

66 Several practical PRNG designs have been proposed, including those in operating systems  
67 such as `/dev/random` [20] for Linux, Yarrow [15] for MacOS/iOS/FreeBSD, and Fortuna [10]  
68 for Windows, and in standards like NIST’s SP 800-90A [2]. Designing secure PRNGs remains  
69 however a complex task, and several flaws have been identified in existing designs (cf. e.g.  
70 [19, 21]).

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

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

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

92 simple case in which the inputs are *independent* and without assuming ideal primitives.

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

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

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

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

## 119 1.1 Our Results

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

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

133 Surprisingly, we show that *no* seedless PRNG can resist premature next attacks, even  
134 if the inputs are sampled independently. In particular, as deterministic extraction without  
135 the seed *is* possible for independent inputs [3], our impossibility is inherently due to the  
136 premature next problem. In more detail, following [8], we parameterize security by two values,  
137  $\gamma^*$ , and  $\beta$ . The goal is to guarantee that *if* the PRNG has obtained  $\gamma^*$  bits of min-entropy

138 within  $T^*$  steps after the last state compromise, then the PRNG will revert to producing  
 139 pseudorandom bits within  $\beta T^*$  steps after the same state compromise. We prove that for any  
 140 choice of  $\gamma^*, \beta$ , there exists an efficient adversary providing  $q \geq \gamma^{*2} \beta^2$  PRNG inputs (each  
 141 with one independent bit of entropy) which violate the PRNG security against premature  
 142 next attacks. Since  $q$  is typically huge, this rules out any reasonable settings of  $\gamma^*$  and  $\beta$ .

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

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

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

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

185 Because of our general impossibility result above, we cannot achieve general  $(\alpha, \beta)$ -security.

186 (We also give direct proof of this fact in the setting of schedulers below.) We then show two  
 187 positive results yielding proven security guarantees for the two schedulers used in the real  
 188 world, by giving meaningful restrictions to the model.

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

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

## 207 2 Preliminaries

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

212 We consider PPT adversaries, in some security parameter  $\lambda$ . All our variables in our  
 213 security definitions will depend on this security parameter.

214 **MIN-ENTROPY.** The *prediction probability* of a random variable  $X$  is  $\text{Pred}(X) := \max_x \mathbb{P}[X = x]$   
 215 and the *min-entropy* is  $H_\infty(X) = -\log(\text{Pred}(X))$ .

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

$$221 \quad 2 \cdot \left| \mathbb{P}[\mathcal{A} \text{ wins}] - 1/2 \right| ,$$

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

---

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

225 apart, i.e.,

$$226 \quad | P[\mathcal{A} = 1|b = 0] - P[\mathcal{A} = 1|b = 1] | .$$

### 227 **3** Impossibility of “Premature Next” Seedless PRNGs

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

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

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

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

#### 261 **3.1 Pseudorandom Number Generators with Input**

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

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

268 ► **Definition 1** (Syntax of PRNGs). A pseudorandom number generator with input (PRNG)  
 269 is a pair of algorithms  $\text{PRNG} = (\text{refresh}, \text{next})$  sharing a  $\mu$ -bit state  $s$ , where

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

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

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

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

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

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

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

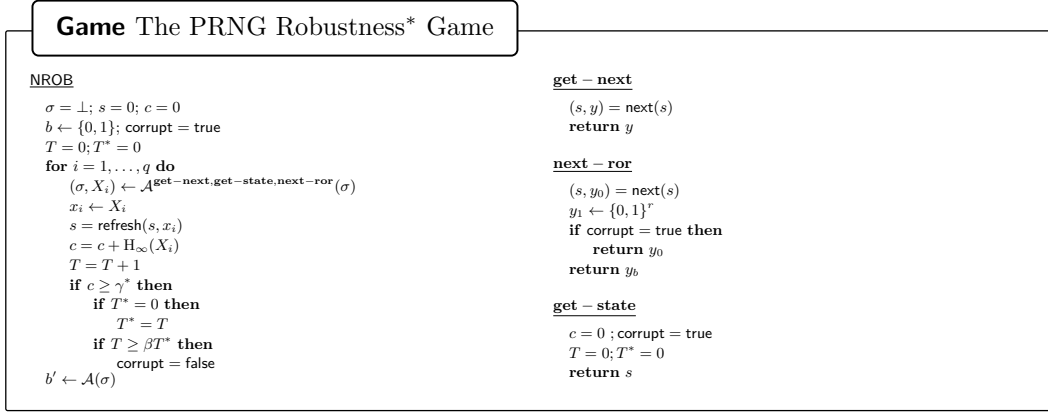
300 ► **Definition 2** (Definition of an Attacker). An attacker  $\mathcal{A}$  is called a  $(q, \tau)$ -attacker if it  
 301 provides at most  $q$  input distributions for **refresh** and runs in time at most  $\tau$ .

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

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

$$\text{Adv}_{\text{PRNG}}^{\text{NROB}}(\mathcal{A}) \leq \epsilon$$





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

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

### 314 3.2 Impossibility Result

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

319 In the remainder of this section, we will work with a function  $f_{\text{PRNG}} : \{0, 1\}^\mu \times \{0, 1\}^n \rightarrow$   
 320  $\{0, 1\}$ , for  $\text{PRNG} = (\text{refresh}, \text{next})$ . This function  $f_{\text{PRNG}}(s, x)$  represents the application of  $n$   
 321 iterated refresh calls, starting from an initial state  $s$  with input  $x_1, \dots, x_n \in \{0, 1\}$ , before  
 322 finally applying next to produce an output bit  $y$ , or more formally:

```

fPRNG(s, x1 || . . . || xn):
  for i = 1 to n
    s = refresh(s, xi)
  (s, y) = next(s)
  return y
  
```

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



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

- 335 ■  $i = 0, z \in \{0, 1\}$ , and  $\mathbb{P}[g(U_n) = z] \geq 0.6$ .
- 336 ■  $1 \leq i \leq n, z \in \{0, 1\}^{n-1}$  and  $g(x_{-i,0}) \neq g(x_{-i,1})$  where  $x_{-i} = z$ .

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

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

342 **ANALYSIS OF FIRST **for** LOOP.** Let us look at trying to determine  $i, z$  such that it satisfies  
 343 the second property. To this end, we will rely on results from graph theory. Specifically, we  
 344 will use the edge isoperimetric inequality for a Hypercube graph [13, §4], which we recall (in  
 345 our context) below.

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

$$352 \quad C \geq k \cdot (n - \log_2 k) \geq k$$

353 However, now we need to determine how many  $\mathbf{u} \in V$  exists such that  $g(\mathbf{u}) = 0$  (or 1). If,  
 354  $0.4 \leq \mathbb{E}[g(U_n)] \leq 0.6$ , then we know that there exists  $0.4 \cdot 2^n$  vectors  $\mathbf{u}$  with  $g(\mathbf{u}) = 0$  and a  
 355 similar number for  $g(\mathbf{u}) = 1$ .

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

$$357 \quad \frac{k}{n \cdot 2^{n-1}} \geq \frac{0.4 \cdot 2^n}{n \cdot 2^{n-1}} = \frac{0.8}{n}$$

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

$$361 \quad \mathbb{P}[g(x_{-i,0}) \neq g(x_{-i,1})] > 1 - 2^{-n}$$

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

$$363 \quad \left(1 - \frac{0.8}{n}\right)^{n^2} \leq e^{-0.8n} < 2^{-n}$$

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

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

```

Algorithm FINDg
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<sup>g</sup>.

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

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

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

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

398 In other words, we have an attacker which can break this scheme, with non-negligible

**Algorithm  $\mathcal{A}$**

```

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

```

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

399 probability, if  $q > \gamma^* \beta^2$ .<sup>2</sup>

### 400 3.3 Towards Positive Results

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

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

412 For such scenarios, we revisit the round-robin Fortuna scheduler, where [8] observe that  
 413 this scheme provably overcomes our impossibility result, by assuming all entropy comes at  
 414 a fixed (but unknown) rate. Instead, in Section 6 we significantly generalize this positive  
 415 result. The idea is to redefine the notion of entropy we use in a way that makes it more  
 416 restrictive than traditional (min-) entropy, but not as restrictive as assuming fixed constant  
 417 rate.<sup>3</sup> Intuitively, our notion of entropy will not allow attacks where the entropy varies too

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

<sup>3</sup>We also note that the results about fast entropy accumulation in the register [5] might justify why

418 widely within a given round-robin (but can change from one round-robin to another) — in a  
 419 sense that the attacker will get almost no credit for high-entropy samples when there is at  
 420 least one low entropy sample within a given round-robin.

## 421 **4 Seedless Scheduler**

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

### 428 **4.1 Syntax of a Scheduler**

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

431 ► **Definition 6** (Syntax of Scheduler). *A  $(k, q)$ -scheduler is a deterministic algorithm SC that*  
 432 *produces  $q$  pairs:  $\{(in_i, out_i)\}_{i=1}^q$  where  $in_i \in [k]$ ,  $out_i \in [k] \cup \{\perp\}$  for  $i = 1, \dots, q$ .*

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

### 437 **4.2 Seedless PRNG, with Premature Next**

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

446 The composition relies on seedless PRNGs which are not secure with premature next.  
 447 These are typically parametrized by just  $\gamma^*$ , which is the minimum entropy needed for the  
 448 PRNG to begin producing pseudorandom outputs (see Figure 5). In essence, these have  
 449  $\alpha = \gamma^*$  and  $\beta = 1$  with a reset of all counters when an adversary invokes **get** – **next**  
 450 **corrupt = true**. The instantiation of this PRNG can be from the work of Coretti *et al.* [4]  
 451 or from the work of Dodis *et al.* [6]. Such a PRNG, secure without premature next and  
 452 parametrized by  $\gamma^*$  is combined with a scheduler. The goal of a scheduler would be to ensure  
 453 that the input, as it arrives, is allocated a particular pool such that:

- 454 ■ With “enough entropy”, a pool is filled, i.e., accumulates  $\gamma^*$  amount of entropy.
- 455 ■ This pool will be emptied within “sufficient time”, to recover from compromise.

---

our new (more restrictive) notion of entropy might be reasonable to expect in practice.

**Construction:** Premature-Next Robust PRNG

<pre> refresh*(x, s̄)   Parse s̄ as (s<sub>0</sub>, ..., s<sub>k-1</sub>, ρ)   in, out ← SC()   s<sub>in</sub> ← refresh(s<sub>in</sub>, x)   if out ≠ ⊥ then     (s<sub>out</sub>, R) ← next(s<sub>out</sub>)     ρ ← ρ ⊕ R   return s̄ = (s<sub>0</sub>, ..., s<sub>k-1</sub>, ρ) </pre>	<pre> next*(s̄)   Parse s̄ as (s<sub>0</sub>, ..., s<sub>k-1</sub>, ρ)   (Y, ρ) ← G(ρ)   return (s̄ = (s<sub>0</sub>, ..., s<sub>k-1</sub>, ρ), Y) </pre>
--	---

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

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

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

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

463 Then, we construct a PRNG with input  $\mathcal{G}(\text{SC}, \{\mathcal{G}_i\}_{i=0}^{k-1}, \mathbf{G}) = (\text{refresh}^*, \text{next}^*)$  as shown in

464 Figure 4, where the scheduler mandates which pool  $\mathcal{G}_{\text{in}}$  to use (via  $\text{refresh}$ ) to accumulate

465 entropy from a new sample, and which pool  $\mathcal{G}_{\text{out}}$  (if any) to “empty” (via  $\text{next}$ ) into the main

466 register  $\rho$  for  $\mathbf{G}$ .

### 467 4.3 Security of a Scheduler

468 We will define different notions of security for a scheduler. As with PRNGs,  $(k, q)$ -scheduler

469 security model is parameterized by two parameters  $\alpha, \beta$ . Informally, it states that if the

470 adversary chooses to provide  $\alpha$  units of fresh entropy (i.e., a sequence of  $w_i$  values that sum

471 up to  $\alpha$ ) within a time  $t \leq q/\beta$ , then we guarantee recovery within time  $\beta \cdot t \leq q$ . Formally,

472 ► **Definition 7** (General Security of Scheduler). *A  $(k, q)$ -scheduler is  $(\alpha, \beta)$ -general-secure if if*

473  *$\forall t_0, t$  such that  $t_1 = t_0 + \beta \cdot t \leq q$ , and  $\forall$  weights  $w_1, \dots, w_q \in [0, 1]$  such that  $\sum_{i=1}^t w_{t_0+i} \geq \alpha$ ,*

474 *the scheme recovers from the compromise in time  $t_0 + \beta t$  where recovery occurs if  $\exists j \in [k]$*

475 *and  $\exists \hat{T} \in [t_0 + 1, t_0 + \beta \cdot t]$  such that:*

- 476 1.  $\text{out}_{t_0+1}, \dots, \text{out}_{\hat{T}-1} \neq j$  (pool  $j$  has not been emptied before time  $\hat{T}$ );
- 477 2.  $\text{out}_{\hat{T}} = j$  (pool  $j$  is emptied at time  $\hat{T}$ ); and
- 478 3. (pool  $j$  has filled)  $\sum_{\substack{t_0 < i \leq \hat{T} \\ \text{in}_i = j}} w_i \geq 1$ .

### 479 4.4 Impossibility Result

480 We can show that, for general security, there exists an impossibility result. Specifically, we

481 will show that for any  $k \in \mathbb{N}$ , there exists a choice of  $q$  such that any  $(k, q)$ -scheduler is

482 not  $(\alpha, \beta)$ -secure. In other words, for a suitable choice of  $q$ , one can break the scheduler to

483 never recover from compromise for any  $\alpha, \beta$ . Note that this is incomparable to the earlier

484 impossibility result discussed in Section 3.2 as this assumes the existence of pools.

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

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

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

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

494 Then, set  $t_0 = \alpha \beta (j^* - 1) - 1$ . After this state compromise, we provide a sequence of  
 495 1s of length  $\alpha$ , which will set  $T^* = \alpha$  and expect recovery in time  $t_0 + \beta T^* = \alpha \beta j^* - 1$ ,  
 496 which is still inside the interval  $I_{j^*}$ . However, we assumed no pool is emptied within  $I_{j^*}$ ,  
 497 so no recovery can be possible.

498 ■  $\forall j, \exists i$  such that  $\text{empty}_i \in I_j$ , meaning at least one pool is emptied within all  $\alpha$  intervals  
 499  $I_j$ .

500 Set  $t_0 = 0$ . Then, for  $j = 1, \dots, \alpha$ , pick *one*  $\ell$  such that  $\text{empty}_\ell \in I_j$ . Set,  $w_\ell$  to be  $1 - \epsilon$   
 501 for some arbitrarily small  $\epsilon$  (remaining weights are 0). At the end of this process, the  
 502 adversary has provided almost  $\alpha$  entropy, but there is no recovery, as all of these entropies  
 503 are completely wasted. By making  $\epsilon$  arbitrarily small, the result follows.

504 ◀ ◀

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

509 **5 Reboot Secure Schedulers**

510 The first relaxation corresponds to the situation when the system is just rebooted, i.e., we  
 511 are at  $t_0 = 0$ . We will call this as the “reboot security” of a scheduler. This corresponds  
 512 to the situation when you just turn on the computer. For this case, we can have a much  
 513 simpler and better RNG, having only one pool. Like Fortuna, this pool is emptied every  $\beta^i$   
 514 steps for gradually increasing values of  $i = 0, 1, 2, \dots$ , where  $\beta$  is a small integer (Windows  
 515 10 uses  $\beta = 3$ ).

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

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

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

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

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

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

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

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

538 Now, let us assume that there is an empty in intervals,  $(\alpha, \alpha\beta], (\alpha\beta, \alpha\beta^2], \dots, (\alpha\beta^{i-1}, \alpha\beta^i]$ .  
 539 We will now show that there needs to be an empty in the interval  $(\alpha\beta^i, \alpha\beta^{i+1}]$ . To this end,  
 540 assume to the contrary. Now, note that the adversary can provide the entropy in such a way  
 541 that every empty in the preceding intervals empties out  $1 - \epsilon$ , without recovering. This is  
 542 similar to the attack detailed in the proof of Theorem 8. Further, if  $t = \alpha\beta^i$ , the scheme has  
 543 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  
 544 to recover in time either. Therefore, there is an empty in the interval  $(\alpha\beta^i, \alpha\beta^{i+1}]$ .  $\triangleleft$

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

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

551  $\blacktriangleright$  **Construction 1 (Reboot Scheme).** *The scheme has  $k = 1$ .  $\text{in}_i = 0$  for  $i = 1, \dots, q$ .*

$$552 \quad \text{out}_i = \begin{cases} 0 & \text{if } i = \beta^j \\ \perp & \text{else} \end{cases}$$

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

554  $\blacktriangleright$  **Theorem 12.** *Construction 1 is  $(\alpha, \beta)$ -reboot secure for  $q = \alpha\beta^\alpha$  (i.e.,  $\alpha \approx \log_\beta q -$   
 555  $\log \log q$ ).*

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

559 Let  $i$  be such that  $\alpha \in (\beta^{i-1}, \beta^i]$ . Now, it is clear that if  $t = \alpha$ , then the empty at  $\beta^i$  will  
 560 ensure recovery from compromise. We can induct similar to the proof of Claim 11 to get that  
 561 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  
 562 of such  $j$ 's which sum up to 1), which is emptied at  $\beta^i$ , thus recovering from compromise.  
 563 Specifically, if we have  $t \in [\beta^{\ell-1}, \beta^\ell)$ , then at each of the preceding  $\ell - 1$  intervals (each with  
 564 an empty),  $\mathcal{A}$  provides  $1 - \epsilon$  entropy, for some arbitrarily small  $\epsilon$ . This gives a total of almost  
 565  $\ell - 1$  entropy across these intervals. Therefore, it follows that the remainder of  $\alpha - \ell + 1 > 1$   
 566 needs to be provided between  $w_{\beta^{\ell-1}}$  and  $w_t$  to hit  $\alpha$  and all of these are emptied at  $\beta^\ell$ ,  
 567 recovering from compromise.  $\blacktriangleleft$   $\blacktriangleleft$



568 **6 Repeat Secure Schedulers**

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

572 **ROUND-ROBIN SCHEDULERS.** Simple round-robin schedulers achieve very good  $\alpha \approx \log_\beta(q)$   
 573 for the special cases when all of the  $w_t$  are equal to some (unknown, adversarially chosen)  
 574 value  $w$ , i.e.,  $w_1 = w_2 = \dots = w_q = w$  and setting the number of pools  $k \approx \log_\beta(q)$  (so 1 or  
 575 2 pools are too little).  $\beta$  is a smaller integer usually 2 or 3 in practice, as in [10, 8]. More  
 576 formally, such schedulers simply set  $\text{in}_t = t \bmod k$ . As for  $\text{out}_t$ , this is set to  $\perp$  inside one  
 577 round (i.e.  $t \bmod k \neq 0$ ). At the the of each round, when  $t = k\ell$ , one looks at the largest  
 578 index  $i \geq 0$  such that  $\beta^i$  divides  $\ell$ . Then out empties the  $i$ -th pool:  $\text{out}_t = i$

579 **► Remark 13.** There is a marginal gain in efficiency when we empty all pools  $\leq i$ , instead of  
 580 just the  $i$ -th pool. In other words,  $\text{out}$  is a set, rather than a single index. However, for our  
 581 analysis below, we will continue to work with the assumption that a single pool is emptied.  
 582 (More generally, we do not make much of an attempt to optimize the parameters that we  
 583 achieve. See [8] for an optimized version of similar construction.)

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

591 **► Definition 14 (Repeating Sequences).**  $\mathbf{w} = (w_1, \dots, w_q)$  with  $0 \leq w_i \leq 1$  is called  $k$ -  
 592 repeating if  $w_{jk+1} = w_{jk+2} = \dots = w_{j(k+k)}$  for  $j = 0, \dots, t-1$  where  $q = k \cdot t$

593 **► Definition 15 (Repeat Security of Scheduler).** A  $(k, q)$ -scheduler is  $(\alpha, \beta, k)$ -repeat-secure  
 594 if  $\forall t_0, t$  such that  $t_1 = t_0 + \beta \cdot t \leq q$ , and  $\forall k$ -repeating weights  $w_1, \dots, w_q \in [0, 1]$  such  
 595 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  
 596 definition of recovery is as defined in Definition 7.

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

**► Definition 16 ( $k$ -Flattening).** Given a sequence  $\mathbf{w} = (w_1, \dots, w_q)$  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  $\mathbf{w}' = (w'_1, \dots, w'_q)$ , where for any round-robin  $j \in \{0, \dots, t-1\}$  and  $i \in \{1 \dots k\}$ , we let

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

599 Intuitively, we change the entropy  $w_j$  to the smallest of  $k$  surrounding entropies inside a  
 600 given round-robin. Of course,  $k = 1$  corresponds to  $w'_t = w_t$ , but we already know that  
 601 1 pool is not enough (as this would give a general scheduler for the unrestricted entropy  
 602 setting). For larger  $k$ , however, the flattened values could be noticeably lower than the  
 603 original. For example, if  $k = 3$  and  $\mathbf{w} = \{1, 1/2, 1/3, 1/4, 1/5, 1/6\}$ , the 3-flattening of  
 604  $\mathbf{w}$  is  $\mathbf{w}' = \{1/3, 1/3, 1/3, 1/6, 1/6, 1/6\}$ . Of course, for a constant rate  $w_1 = \dots = w_q = w$ ,  
 605  $k$ -flattening does not change anything, which explains why our results below naturally  
 606 generalize the constant-rate analysis from the work of Dodis *et al.* [8].

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

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

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

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

622 ■  $\text{in}_i = i \bmod k$   
 623 ■  $\text{out}_i = \begin{cases} \perp & \text{if } i \bmod k \neq 0 \\ j & \text{if } i = k\ell \end{cases}$  where  $j \geq 0$  is the largest  $j$  such that  $\ell \bmod b^j = 0$  for  
 624  $i = k\ell$

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

► **Theorem 17.** For any integers  $b \geq 2$  and  $k \geq 1$ , Construction 2 is  $(\alpha, \beta, k)$ -repeat-secure  
 for

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

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

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

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

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

$$642 \sum_{\substack{t_0 < i \leq \widehat{T} \\ \text{in}_i = j}} w_i \geq \sum_{t_0 < i \leq \widehat{T}} \sum_{\substack{t_0 < i \leq t_0+t \\ \text{in}_i = j}} w_i \geq \sum_{\substack{t'_0 < i \leq t'_0+t' \\ \text{in}_i = j}} w_i = \frac{1}{k} \cdot \sum_{t'_0 < i \leq t'_0+t'} w_i,$$

## 9:18 On Seedless PRNGs and Premature Next

643 where  $t'_0 := \lceil t_0/k \rceil k \geq t_0$  and  $t' := \lfloor t/k \rfloor k \leq t$ . And, since  $w_i \leq 1$ , we trivially have that

$$644 \quad \sum_{t'_0 < i \leq t'_0 + t'} w_i \geq \sum_{t_0 < i \leq t_0 + t} w_i - 2k + 2 \geq \alpha - 2k + 2.$$

645 Therefore,

$$646 \quad \sum_{\substack{t_0 < i \leq t_0 + t \\ \text{in}_i = j}} w_i \geq \frac{\alpha}{k} - 2 + 2/k \geq 1,$$

647 as needed. ◀

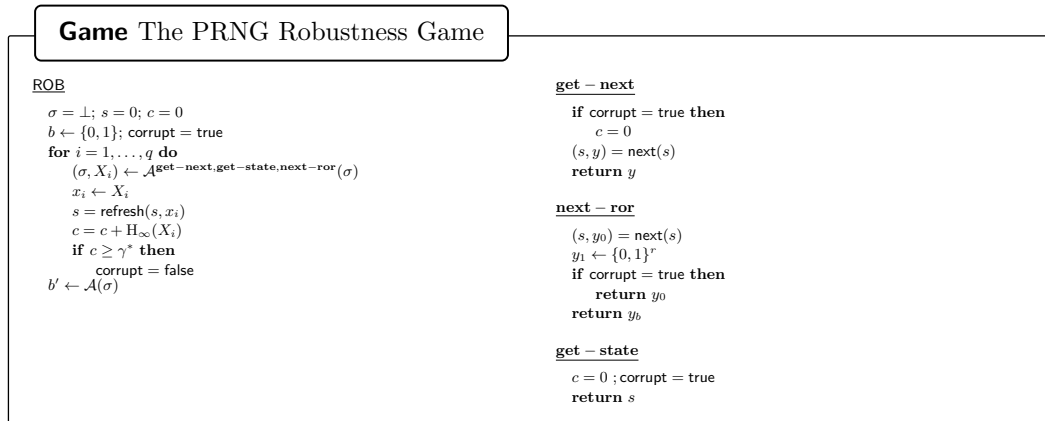
648 Notice, this result explains how the recovery factor  $\beta$  shrinks very quickly as we increase  
649 the number of pools  $k$ , starting with (roughly)  $q$  all the way down to being a constant. In  
650 particular,  $\beta$  becomes constant once the number of pools becomes logarithmic in 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

651 Moreover, up to constant factors in  $\alpha$  and  $\beta$  (which, again, we do not attempt to  
652 optimize), Theorem 17 is tight. In particular, [8, Proposition 1] proved that even in  
653 the “constant-rate” case of  $q$ -repeating weights, no scheduler can be  $(\alpha, \beta)$ -secure with  
654  $\alpha\beta \leq \log_e q - \log_e \log_e q - 1$ . And our scheduler matches this bound (up to a constant factor)  
655 when  $b = O(1)$  and  $k = O(\log q)$ .

## References

- 656 1 Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with  
657 applications to /dev/random. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels,  
658 editors, *ACM CCS 2005*, pages 203–212, Alexandria, Virginia, USA, November 7–11, 2005.  
659 ACM Press. doi:10.1145/1102120.1102148.
- 660 2 Elaine Barker and John Kelsey. Recommendation for random number generation using  
661 deterministic random bit generators. NIST Special Publication 800-90A, 2012.
- 662 3 Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and  
663 probabilistic communication complexity. *SIAM J. Comput.*, 17(2):230–261, 1988. doi:  
664 10.1137/0217015.
- 665 4 Sandro Coretti, Yevgeniy Dodis, Harish Karthikeyan, and Stefano Tessaro. Seedless Fruit is  
666 the sweetest: Random number generation, revisited. In Alexandra Boldyreva and Daniele  
667 Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 205–234, Santa  
668 Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany. doi:10.1007/  
669 978-3-030-26948-7\_8.
- 670 5 Yevgeniy Dodis, Siyao Guo, Noah Stephens-Davidowitz, and Zhiye Xie. No time to hash: On  
671 super-efficient entropy accumulation. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021,*  
672 *Part IV*, volume 12828 of *LNCS*, pages 548–576, Virtual Event, August 16–20, 2021. Springer,  
673 Heidelberg, Germany. doi:10.1007/978-3-030-84259-8\_19.
- 674 6 Yevgeniy Dodis, Siyao Guo, Noah Stephens-Davidowitz, and Zhiye Xie. Online linear extractors  
675 for independent sources. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic*  
676 *Cryptography, ITC 2021, July 23-26, 2021, Virtual Conference*, volume 199 of *LIPICs*, pages  
677 14:1–14:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.  
678 ITC.2021.14.
- 679 7 Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs.  
680 Security analysis of pseudo-random number generators with input: /dev/random is not robust.  
681 In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages  
682 647–658, Berlin, Germany, November 4–8, 2013. ACM Press. doi:10.1145/2508859.2516653.
- 683 8 Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your  
684 entropy and have it too: Optimal recovery strategies for compromised rngs. *Algorithmica*,  
685 79(4):1196–1232, 2017. doi:10.1007/s00453-016-0239-3.
- 686 9 Niels Ferguson. The windows 10 random number generation infrastructure, Oct 2019. URL:  
687 <https://aka.ms/win10rng>.
- 688 10 Niels Ferguson and Bruce Schneier. *Practical cryptography*. Wiley, 2003.
- 689 11 Peter Gazi and Stefano Tessaro. Provably robust sponge-based PRNGs and KDFs. In  
690 Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665  
691 of *LNCS*, pages 87–116, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.  
692 doi:10.1007/978-3-662-49890-3\_4.
- 693 12 Viet Tung Hoang and Yaobin Shen. Security analysis of NIST CTR-DRBG. In Daniele  
694 Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*,  
695 pages 218–247, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.  
696 doi:10.1007/978-3-030-56784-2\_8.
- 697 13 Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications.  
698 *BULL. AMER. MATH. SOC.*, 43(4):439–561, 2006.
- 699 14 Daniel Hutchinson. A robust and sponge-like PRNG with improved efficiency. In Roberto  
700 Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 381–398,  
701 St. John’s, NL, Canada, August 10–12, 2016. Springer, Heidelberg, Germany. doi:10.1007/  
702 978-3-319-69453-5\_21.
- 703 15 John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and  
704 analysis of the yarrow cryptographic pseudorandom number generator. In *In Sixth Annual*  
705 *Workshop on Selected Areas in Cryptography*, pages 13–33. Springer, 1999.
- 706

- 707 **16** John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on  
708 pseudorandom number generators. In Serge Vaudenay, editor, *FSE'98*, volume 1372 of  
709 *LNCS*, pages 168–188, Paris, France, March 23–25, 1998. Springer, Heidelberg, Germany.  
710 doi:10.1007/3-540-69710-1\_12.
- 711 **17** Noam Nisan and David Zuckerman. Randomness is linear in space. *J. Comput. Syst. Sci.*,  
712 52(1):43–52, 1996. doi:10.1006/jcss.1996.0004.
- 713 **18** Amit Sahai and Salil Vadhan. A complete problem for statistical zero knowledge. *Journal of*  
714 *the ACM*, 50(2):196–249, 2003. Extended abstract in FOCS '97.
- 715 **19** Thomas Shrimpton and R. Seth Terashima. A provable-security analysis of Intel's secure  
716 key RNG. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*,  
717 volume 9056 of *LNCS*, pages 77–100, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg,  
718 Germany. doi:10.1007/978-3-662-46800-5\_4.
- 719 **20** Wikipedia contributors. /dev/random — Wikipedia, the free encyclopedia, 2021. [On-  
720 line; accessed 9-January-2022]. URL: [https://en.wikipedia.org/w/index.php?title=/dev/  
721 random&oldid=1056079736](https://en.wikipedia.org/w/index.php?title=/dev/random&oldid=1056079736).
- 722 **21** Joanne Woodage and Dan Shumow. An analysis of NIST SP 800-90A. In Yuval Ishai  
723 and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages  
724 151–180, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany. doi:  
725 10.1007/978-3-030-17656-3\_6.



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

## 726 **A** PRNG Robustness, without Premature Next

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