How to Eat Your Entropy and Have it Too — Optimal Recovery Strategies for Compromised RNGs

Yevgeniy Dodis[1⋆], Adi Shamir[2], Noah Stephens-Davidowitz[1], and Daniel Wichs[3⋆⋆]

[1] Dept. of Computer Science, New York University. { dodis@cs.nyu.edu, noahsd@gmail.com }
[2] Dept. of Computer Science and Applied Mathematics, Weizmann Institute. adi.shamir@weizmann.ac.il
[3] Dept. of Computer Science, Northeastern University. wichs@ccs.neu.edu

Abstract. Random number generators (RNGs) play a crucial role in many cryptographic schemes and protocols, but their security proof usually assumes that their internal state is initialized with truly random seeds and remains secret at all times. However, in many practical situations these are unrealistic assumptions: The seed is often gathered after a reset/reboot from low entropy external events such as the timing of manual key presses, and the state can be compromised at unknown points in time via side channels or penetration attacks. The usual remedy (used by all the major operating systems, including Windows, Linux, FreeBSD, MacOS, iOS, etc.) is to periodically replenish the internal state through an auxiliary input with additional randomness harvested from the environment. However, recovering from such attacks in a provably correct and computationally optimal way had remained an unsolved challenge so far.

In this paper we formalize the problem of designing an efficient recovery mechanism from state compromise, by considering it as an online optimization problem. If we knew the timing of the last compromise and the amount of entropy gathered since then, we could stop producing any outputs until the state becomes truly random again. However, our challenge is to recover within a time proportional to this optimal solution even in the hardest (and most realistic) case in which (a) we know nothing about the timing of the last state compromise, and the amount of new entropy injected since then into the state, and (b) any premature production of outputs leads to the total loss of all the added entropy *used by the RNG*, since the attacker can use brute force to enumerate all the possible low-entropy states. In other words, the challenge is to develop recovery mechanisms which are guaranteed to save the day as quickly as possible after a compromise we are not even aware of. The dilemma that we face is that any entropy used prematurely will be lost, and any entropy which is kept unused will delay the recovery.

After developing our formal definitional framework for RNGs with inputs, we show how to construct a nearly optimal RNG which is secure in our model. Our technique is inspired by the design of the Fortuna RNG (which is a heuristic RNG construction that is currently used by Windows and comes without any formal analysis), but we non-trivially adapt it to our much stronger adversarial setting. Along the way, our formal treatment of Fortuna enables us to improve its entropy efficiency by almost a factor of two, and to show that our improved construction is essentially tight, by proving a rigorous lower bound on the possible efficiency of any recovery mechanism in our very general model of the problem.

1 Introduction

Randomness is essential in many facets of cryptography, from the generation of long-term cryptographic keys, to sampling local randomness for encryption, zero-knowledge proofs, and many other randomized cryptographic primitives. As a useful abstraction, designers of such cryptographic schemes assume a source of (nearly) uniform, unbiased, and independent random bits of arbitrary length. In practice, however, this theoretical abstraction is realized by means of a *Random Number Generator* (RNG), whose goal is to quickly accumulate entropy from various physical sources in the environment (such as keyboard presses or mouse movement) and then convert it into the required source of (pseudo) random bits. We notice that a highly desired (but, alas, rarely achieved) property of such RNGs is their ability to quickly recover from various

⋆ Research partially supported by gifts from VMware Labs and Google, and NSF grants 1319051, 1314568, 1065288, 1017471.
⋆⋆ Research partially supported by gift from Google and NSF grants 1347350, 1314722.

forms of *state compromise*, in which the current state S of the RNG becomes known to the attacker, either due to a successful penetration attack, or via side channel leakage, or simply due to insufficient randomness in the initial state. This means that the state S of practical RNGs should be periodically refreshed using the above-mentioned physical sources of randomness I. In contrast, the simpler and much better-understood theoretical model of pseudorandom generators (PRGs) does not allow the state to be refreshed after its initialization. To emphasize this distinction, we will sometimes call our notion an "RNG with input", and notice that virtually all modern operating systems come equipped with such an RNG with input; e.g., `/dev/random` [20] for Linux, Yarrow [13] for MacOs/iOS/FreeBSD and Fortuna [9] for Windows [8].

Unfortunately, despite the fact that they are widely used and often referred to in various standards [2, 7, 12, 15], RNGs with input have received comparatively little attention from theoreticians. The two notable exceptions are the works of Barak and Halevi [1] and Dodis et al. [5]. The pioneering work of [1] emphasized the importance of rigorous analysis of RNGs with input and laid their first theoretical foundations. However, as pointed out by [5], the extremely clean and elegant security model of [1] ignores the "heart and soul" issue of most real-world RNGs with input, namely, their ability to gradually "accumulate" many low-entropy inputs I into the state S at the same time that they lose entropy due to premature use. In particular, [5] showed that the construction of [1] (proven secure in their model) may always fail to recover from state compromise when the entropy of each input $I_1, \ldots, I_q$ is sufficiently small, *even for arbitrarily large* q.

Motivated by these considerations, Dodis et al. [5] defined an improved security model for RNGs with input, which explicitly guaranteed eventual recovery from any state compromise, provided that the *collective* fresh entropy of inputs $I_1, \ldots, I_q$ crosses some security threshold γ^*, *irrespective of the entropies of individual inputs* I_j. In particular, they demonstrated that Linux's `/dev/random` does not satisfy their stronger notion of *robustness* (for similar reasons as the construction of [1]), and then constructed a simple scheme which is provably robust in this model. However, as we explain below, their robustness model did not address the issue of efficiency of the recovery mechanism when the RNG is being *continuously used* after the compromise.

The Premature Next Problem. In this paper, we extend the model of [5] to address some additional desirable security properties of RNGs with input not captured by this model. The main such property is resilience to the "*premature next* attack". This general attack, first explicitly mentioned by Kelsey, Schneier, Wagner, and Hall [14], is applicable in situations in which the RNG state S has accumulated an insufficient amount of entropy e (which is very common in bootup situations) and then must produce some outputs R via legitimate "next" calls in order to generate various system keys. Not only is this R not fully random (which is expected), but now the attacker can potentially use R to recover the current state S by brute force, effectively "emptying" the e bits of entropy that S accumulated so far. Applied iteratively, this simple attack, when feasible, can prevent the system from ever recovering from compromise, irrespective of the total amount of fresh entropy injected into the system since the last compromise.

At first, it might appear that the only way to prevent this attack is by discovering a sound way to estimate the current entropy in the state and to use this estimate to block the premature next calls. This is essentially the approach taken by Linux's `/dev/random` and many other RNGs with input. Unfortunately, sound entropy estimation is hard or even infeasible [9, 19] (e.g., [5] showed simple ways to completely fool Linux's entropy estimator). This seems to suggest that the modeling of RNGs with input should consider each premature next call as a full state compromise, and this is the highly conservative approach taken by [5] (which we will fix in this work).

Fortuna. Fortunately, the conclusion above is overly pessimistic. In fact, the solution idea already comes from two very popular RNGs mentioned above, whose designs were heavily affected by the desire to overcome the premature next problem: Yarrow (designed by Schneier, Kelsey and Ferguson [13] and used by MacOS/iOS/FreeBSD), and its refinement Fortuna (subsequently designed by Ferguson and Schneier [9] and used by Windows [8]). The simple but brilliant idea of these works is to partition the incoming entropy into multiple entropy "pools" and then to cleverly use these pools at vastly different rates when producing outputs, in order to guarantee that at least one pool will eventually accumulate enough entropy to guarantee security before it is "prematurely emptied" by a next call. (See Section 4 for more details.)

Ferguson and Schneier provide good security intuition for their Fortuna "pool scheduler" construction, assuming that all the RNG inputs $I_1, \ldots, I_q$ have the same (unknown) entropy and that each of the pools can losslessly accumulate all the entropy that it gets. (They suggest using iterated hashing with a cryptographic hash function as a heuristic way to achieve this.) In particular, if q is the upper bound on the number of inputs, they suggest that one can make the number of pools $P = \log_2 q$, and recover from state compromise (with premature next!) at the loss of a factor $O(\log q)$ in the amount of fresh entropy needed.

Our Main Result. Inspired by the idea of Fortuna, we formally extend the prior RNG robustness notion of [5] to *robustness against premature next.* Unlike Ferguson and Schneier, we do so without making any restrictive assumptions such as requiring that the entropy of all the inputs I_j be constant. (Indeed, these entropies can be adversarily chosen, as in the model of [5], and can be *unknown* to the RNG.) Also, in our formal and general security model, we do not assume ideal entropy accumulation or inherently rely on cryptographic hash functions. In fact, our model is syntactically very similar to the prior RNG model of [5], except: (1) a premature next call is not considered an unrecoverable state corruption, but (2) in addition to the (old) "entropy penalty" parameter γ^*, there is a (new) "time penalty" parameter $\beta \geq 1$, measuring how long it will take to recover from state compromise relative to the optimal recovery time needed to receive γ^* bits of fresh entropy. (See Figures 2 and 3.)

To summarize, our model formalizes the problem of designing an efficient recovery mechanism from state compromise as an online optimization problem. If we knew the timing of the last compromise and the amount of entropy gathered since then, we could stop producing any outputs until the state becomes truly random again. However, our challenge is to recover within a time proportional to this optimal solution even in the hardest (and most realistic) case in which (a) we know nothing about the timing of the last state compromise, and the amount of new entropy injected since then into the state, and (b) any premature production of outputs leads to the total loss of all the added entropy *used by the RNG*, since the attacker can use brute force to enumerate all the possible low-entropy states. In other words, the challenge is to develop recovery mechanisms which are guaranteed to save the day as quickly as possible after a compromise we are not even aware of. The dilemma that we face is that *any entropy used prematurely will be lost, and any entropy which is kept unused will delay the recovery.*

After extending our model to handle premature next calls, we define the generalized Fortuna construction, which is provably robust against premature next. Although heavily inspired by actual Fortuna, the syntax of our construction is noticeably different (See Figure 5), since we prove it secure in a stronger model and without any idealized assumptions (like perfect entropy accumulation, which, as demonstrated by the attacks in [5], is not a trivial thing to sweep under the rug). In fact, to obtain our construction, we: (a) abstract out a rigorous security notion of a (pool) *scheduler*; (b) show a formal composition theorem (Theorem 2) stating that a secure scheduler can be composed with any robust RNG in the prior model of [5] to achieve security against premature next; (c) obtain our final RNG by using the provably secure RNG of [5] and a Fortuna-like scheduler (proven secure in our significantly stronger model). In particular, the resulting RNG is secure in the standard model, and only uses the existence of standard PRGs as its sole computational assumption.

Constant-Rate RNGs. In Section 5.4, we consider the actual constants involved in our construction, and show that under a reasonable setting or parameters, our RNG will recover from compromise in $\beta = 4$ times the number of steps it takes to get 20 to 30 kB of fresh entropy. While these numbers are a bit high, they are also obtained in an extremely strong adversarial model. In contrast, remember that Ferguson and Schneier informally analyzed the security of Fortuna in a much simpler case in which entropy drips in at a constant rate. While restrictive, in Section 6 we also look at the security of generalized Fortuna (with a better specialized scheduler) in this model, as it could be useful in some practical scenarios and allow for a more direct comparison with the original Fortuna. In this simpler constant entropy dripping rate, we estimate that our RNG (with standard security parameters) will recover from a complete compromise immediately after it gets about 2 to 3 kB of entropy (see Section 6.2), which is comparable to [9]'s (corrected) claim, but without assuming ideal entropy accumulation into the state. In fact, our optimized constant-rate scheduler beats the original Fortuna's scheduler by almost a factor of 2 in terms of entropy efficiency.

Rate Lower Bound. We also show that any "Fortuna-like construction" (which tries to collect entropy in multiple pools and cleverly utilize them with an arbitrary scheduler) must lose at least a factor $\Omega(\log q)$ in its "entropy efficiency", even in the case where all inputs I_j have an (unknown) *constant-rate* entropy. This suggests that the original scheduler of Fortuna (which used $\log q$ pools which evenly divide the entropy among them) is asymptotically optimal in the constant-rate case (as is our improved version).

Semi-Adaptive Set-Refresh. As a final result, we make progress in addressing another important limitation of the model of Dodis et al. [5] (and our direct extension of the current model that handles premature nexts). Deferring technical details to Section 7, in that model the attacker $\mathcal{A}$ had very limited opportunities to adaptively influence the samples produced by another adversarial quantity, called the *distribution sampler* $\mathcal{D}$. As explained in there and in [5], *some* assumption of this kind is necessary to avoid impossibility results, but it does limit the applicability of the model to some real-world situations. As the initial step to removing this limitation, in Section 7.1 we introduce the "semi-adaptive set-refresh" model and show that both the original RNG of [5] and our new RNG are provably secure in this more realistic adversarial model.

Other Related Work. As we mentioned, there is very little literature focusing on the design and analysis of RNGs with inputs in the standard model. In addition to [1,5], some analysis of the Linux RNG was done by Lacharme, Röck, Strubel and Videau [16]. On the other hand, many works showed devastating attacks on various cryptographic schemes when using weak randomness; some notable examples include [4,6,10,11, 14,17,18].

2 Preliminaries

Entropy. For a discrete distribution X, we denote its *min-entropy* by $\mathbf{H}_\infty(X) = \min_x\{-\log \Pr[X = x]\}$. We also define worst-case min-entropy of X conditioned on another random variable Z by in the following conservative way: $\mathbf{H}_\infty(X|Z) = -\log([\max_{x,z} \Pr[X = x|Z = z]])$. We use this definition instead of the usual one so that it satisfies the following relation, which is called the "chain rule": $\mathbf{H}_\infty(X, Z) - \mathbf{H}_\infty(Z) \geq \mathbf{H}_\infty(X|Z)$.

Pseudorandom Functions and Generators. We say that a function $\mathbf{F} : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}^m$ is a (deterministic) $(t, q_\mathbf{F}, \varepsilon)$-*pseudorandom function* (PRF) if no adversary running in time t and making $q_\mathbf{F}$ oracle queries to $\mathbf{F}(\mathsf{key}, \cdot)$ can distinguish between $\mathbf{F}(\mathsf{key}, \cdot)$ and a random function with probability greater than ε when $\mathsf{key} \xleftarrow{\$} \{0,1\}^\ell$. We say that a function $\mathbf{G} : \{0,1\}^m \to \{0,1\}^n$ is a (deterministic) (t, ε)-*pseudorandom generator* (PRG) if no adversary running in time t can distinguish between $\mathbf{G}(\mathsf{seed})$ and uniformly random bits with probability greater than ε when $\mathsf{seed} \xleftarrow{\$} \{0,1\}^m$.

Game Playing Framework. For our security definitions and proofs we use the code-based game-playing framework of [3]. A game GAME has an $\mathsf{initialize}$ procedure, procedures to respond to adversary oracle queries, and a $\mathsf{finalize}$ procedure. A game GAME is executed with an adversary $\mathcal{A}$ as follows: First, $\mathsf{initialize}$ executes, and its outputs are the inputs to $\mathcal{A}$. Then $\mathcal{A}$ executes, its oracle queries being answered by the corresponding procedures of GAME. When $\mathcal{A}$ terminates, its output becomes the input to the $\mathsf{finalize}$ procedure. The output of the latter is called the output of the game, and we let $\mathsf{GAME}^\mathcal{A} \Rightarrow y$ denote the event that this game output takes value y. $\mathcal{A}^{\mathsf{GAME}}$ denotes the output of the adversary and $\mathsf{Adv}_\mathcal{A}^{\mathsf{GAME}} = 2 \times \Pr[\mathsf{GAME}^\mathcal{A} \Rightarrow 1] - 1$. Our convention is that Boolean flags are assumed initialized to false and that the running time of the adversary $\mathcal{A}$ is defined as the total running time of the game with the adversary in expectation, including the procedures of the game.

3 RNG with Input: Modeling and Security

In this section we present formal modeling and security definitions for RNGs with input, largely following [5].

Definition 1 (RNG with input). *An RNG with input is a triple of algorithms* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *and a triple* $(n, \ell, p) \in \mathbb{N}^3$ *where* n *is the* state length, ℓ *is the output length and* p *is the input length of* $\mathcal{G}$*:*

- setup*: a probabilistic algorithm that outputs some public parameters* seed *for the generator.*
- refresh*: a deterministic algorithm that, given* seed*, a state $S \in \{0,1\}^n$ and an input $I \in \{0,1\}^p$, outputs a new state $S' =$* refresh(seed*, $S, I) \in \{0,1\}^n$.*
- next*: a deterministic algorithm that, given* seed *and a state $S \in \{0,1\}^n$, outputs a pair $(S', R) =$* next(seed*, S) where $S' \in \{0,1\}^n$ is the new state and $R \in \{0,1\}^\ell$ is the output.*

Before moving to defining our security notions, we notice that there are two adversarial entities we need to worry about: the *adversary* $\mathcal{A}$ whose task is (intuitively) to distinguish the outputs of the RNG from random, and the *distribution sampler* $\mathcal{D}$ whose task is to produce inputs $I_1, I_2, \ldots$, which have high entropy *collectively*, but somehow help $\mathcal{A}$ in breaking the security of the RNG. In other words, the distribution sampler models potentially adversarial environment (or "nature") where our RNG is forced to operate.

3.1 Distribution Sampler

The distribution sampler $\mathcal{D}$ is a *stateful and probabilistic* algorithm which, given the current state σ, outputs a tuple (σ', I, γ, z) where: (a) σ' is the new state for $\mathcal{D}$; (b) $I \in \{0,1\}^p$ is the next input for the refresh algorithm; (c) γ is some *fresh entropy estimation* of I, as discussed below; (d) z is the *leakage* about I given to the attacker $\mathcal{A}$. We denote by $q_{\mathcal{D}}$ the upper bound on number of executions of $\mathcal{D}$ in our security games, and say that $\mathcal{D}$ is *legitimate* if $\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_{\mathcal{D}}}, z_1, \ldots, z_{q_{\mathcal{D}}}, \gamma_0, \ldots, \gamma_{q_{\mathcal{D}}}) \geq \gamma_j$ for all $j \in \{1, \ldots, q_{\mathcal{D}}\}$ where $(\sigma_i, I_i, \gamma_i, z_i) = \mathcal{D}(\sigma_{i-1})$ for $i \in \{1, \ldots, q_{\mathcal{D}}\}$ and $\sigma_0 = 0$.[1]

We explain [5]'s reason for explicitly requiring $\mathcal{D}$ to output the entropy estimate γ_j. Most complex RNGs are worried about the situation where the system might enter a prolonged state where no new entropy is inserted in the system. Correspondingly, such RNGs typically include some ad hoc *entropy estimation procedure* E whose goal is to block the RNG from outputting output value R_j until the state has accumulated enough entropy γ^* (for some entropy threshold γ^*). Unfortunately, it is well-known that even approximating the entropy of a given distribution is a computationally hard problem [19]. This means that if we require our RNG $\mathcal{G}$ to explicitly come up with such a procedure E, we are bound to either place some significant restrictions (or assumptions) on $\mathcal{D}$, or rely on some hoc and non standard assumptions. Indeed, [5] demonstrate some attacks on the entropy estimation of the Linux RNG, illustrating how hard (or, perhaps, impossible?) it is to design a sound entropy estimation procedure E. Finally, we observe that the design of E is anyway completely *independent* of the mathematics of the actual refresh and next procedures, meaning that the latter can and *should* be evaluated independently of the "accuracy" of E.

Motivated by these considerations, [5] do not insist on any "entropy estimation" procedure as a mandatory part of the RNG design. Instead, they place the burden of entropy estimations on $\mathcal{D}$ *itself*. Equivalently, we can think that the entropy estimations γ_j come from the entropy estimation procedure E (which is now "merged" with $\mathcal{D}$), but only provide security assuming that E is correct in this estimation (which we know is hard in practice, and motivates future work in this direction).

However, we stress that: (a) the entropy estimates γ_j will only be used in our security definitions, but not in any of the actual RNG operations (which will only use the input I returned by $\mathcal{D}$); (b) we do not insist that a legitimate $\mathcal{D}$ can perfectly estimate the fresh entropy of its next sample I_j, but only provide a *lower bound* γ_j that is legitimate. For example, $\mathcal{D}$ is free to set $\gamma_j = 0$ as many times as it wants and, in this case, can even choose to leak the entire I_j to $\mathcal{A}$ via the leakage z_j! More generally, we allow $\mathcal{D}$ to inject new entropy γ_j as slowly (and maliciously!) as it wants, but will only require security when the counter c keeping track of the current "fresh" entropy in the system[2] crosses some entropy threshold γ^* (since otherwise we have "no reason" to expect any security).

[1] Since conditional min-entropy is defined in the worst-case manner, the value γ_j in the bound below should not be viewed as a random variable, but rather as an arbitrary fixing of this random variable.

[2] Intuitively, "fresh" refers to the new entropy in the system since the last state compromise.

3.2 Security Notions

We define the game $\mathsf{ROB}(\gamma^*)$ in our game framework. We show the $\mathsf{initialize}$ and $\mathsf{finalize}$ procedures for $\mathsf{ROB}(\gamma^*)$ in Figure 1. The attacker's goal is to guess the correct value b picked in the initialize procedure with access to several oracles, shown in Figure 2. Dodis et al. define the notion of *robustness* for an RNG with input [5]. In particular, they define the parametrized security game $\mathsf{ROB}(\gamma^*)$ where γ^* is a measure of the "fresh" entropy in the system when security should be expected. Intuitively, in this game $\mathcal{A}$ is able to view or change the state of the RNG ($\mathsf{get\text{-}state}$ and $\mathsf{set\text{-}state}$), to see output from it ($\mathsf{get\text{-}next}$), and to update it with a sample I_j from $\mathcal{D}$ ($\mathcal{D}$-$\mathsf{refresh}$). In particular, notice that the $\mathcal{D}$-$\mathsf{refresh}$ oracle keeps track of the fresh entropy in the system and declares the RNG to no longer be corrupted only when the fresh entropy c is greater than γ^*. (We stress again that the entropy estimates γ_i and the counter c are not available to the RNG.) Intuitively, $\mathcal{A}$ wins if the RNG is not corrupted and he correctly distinguishes the output of the RNG from uniformly random bits.

proc. initialize
seed $\xleftarrow{\$}$ setup; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$; $c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{\$} \{0,1\}$
`OUTPUT` seed

proc. finalize(b^*)
`IF` $b = b^*$ `RETURN` 1
`ELSE RETURN` 0

Fig. 1: Procedures $\mathsf{initialize}$ and $\mathsf{finalize}$ for $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$

proc. $\mathcal{D}$-refresh
$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
$S \leftarrow \mathsf{refresh}(S, I)$
$c \leftarrow c + \gamma$
`IF` $c \geq \gamma^*$,
  corrupt $\leftarrow$ false
`OUTPUT` (γ, z)

proc. next-ror
$(S, R_0) \leftarrow \mathsf{next}(S)$
$R_1 \xleftarrow{\$} \{0,1\}^\ell$
`IF` corrupt = true,
  $c \leftarrow 0$
  `RETURN` R_0
`ELSE OUTPUT` R_b

proc. get-next
$(S, R) \leftarrow \mathsf{next}(S)$
`IF` corrupt = true,
  $c \leftarrow 0$
`OUTPUT` R

proc. get-state
$c \leftarrow 0$; corrupt $\leftarrow$ true
`OUTPUT` S

proc. set-state(S^*)
$c \leftarrow 0$; corrupt $\leftarrow$ true
$S \leftarrow S^*$

Fig. 2: Procedures in $\mathsf{ROB}(\gamma^*)$ for $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$

Definition 2 (Security of RNG with input). *A pseudorandom number generator with input* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *is called* $((t, q_\mathcal{D}, q_R, q_S), \gamma^*, \varepsilon)$-robust *if for any attacker* $\mathcal{A}$ *running in time at most* t*, making at most* $q_\mathcal{D}$ *calls to* $\mathcal{D}$-$\mathsf{refresh}$*,* q_R *calls to* $\mathsf{next\text{-}ror}/\mathsf{get\text{-}next}$ *and* q_S *calls to* $\mathsf{get\text{-}state}/\mathsf{set\text{-}state}$*, and any legitimate distribution sampler* $\mathcal{D}$ *inside the* $\mathcal{D}$-$\mathsf{refresh}$ *procedure, the advantage of* $\mathcal{A}$ *in game* $\mathsf{ROB}(\gamma^*)$ *is at most* ε*.*

Notice that in $\mathsf{ROB}(\gamma^*)$, if $\mathcal{A}$ calls $\mathsf{get\text{-}next}$ when the RNG is still corrupted, this is a "premature" $\mathsf{get\text{-}next}$ and the entropy counter c is reset to 0. Intuitively, [5] treats information "leaked" from an insecure RNG as a total compromise. We modify their security definition and define the notion of *robustness against premature next* with the corresponding security game $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$. Our modified game $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ has identical $\mathsf{initialize}$ and $\mathsf{finalize}$ procedures to [5]'s $\mathsf{ROB}(\gamma^*)$ (Figure 1). Figure 3 shows the new oracle queries. The differences with $\mathsf{ROB}(\gamma^*)$ are highlighted for clarity.

In our modified game, "premature" $\mathsf{get\text{-}next}$ calls do not reset the entropy counter. We pay a price for this that is represented by the parameter $\beta \geq 1$. In particular, in our modified game, the game does not immediately declare the state to be uncorrupted when the entropy counter c passes the threshold γ^*. Instead, the game keeps a counter T that records the number of calls to $\mathcal{D}$-$\mathsf{refresh}$ since the last $\mathsf{set\text{-}state}$ or $\mathsf{get\text{-}state}$ (or the start of the game). When c passes γ^*, it sets $T^* \leftarrow T$ and the state becomes uncorrupted only after $T \geq \beta T^*$ (of course, provided $\mathcal{A}$ made no additional calls to $\mathsf{get\text{-}state}$ or $\mathsf{set\text{-}state}$). In particular, while we allow extra time for recovery, notice that we do *not* require any additional entropy from the distribution sampler $\mathcal{D}$.

Intuitively, we allow $\mathcal{A}$ to receive output from a (possibly corrupted) RNG and, therefore, to potentially learn information about the state of the RNG without any "penalty". However, we allow the RNG additional time to "mix the fresh entropy" received from $\mathcal{D}$, proportional to the amount of time T^* that it took to get the required fresh entropy γ^* since the last compromise.

As a final subtlety, we set a maximum $\gamma_{\max}$ on the amount that the entropy counter can be increased from one $\mathcal{D}$-refresh call. This might seem strange, since it is not obvious how receiving too much entropy at once could be a problem. However, $\gamma_{\max}$ will prove quite useful in the analysis of our construction. Intuitively, this is because it is harder to "mix" entropy if it comes too quickly. Of course $\gamma_{\max}$ is bounded by the length of the input p, but in practice we often expect it to be substantially lower. In such cases, we are able to prove much better performance for our RNG construction, *even if* $\gamma_{\max}$ *is unknown to the RNG.* In addition, we get very slightly better results if some upper bound on $\gamma_{\max}$ is incorporated into the construction.

proc. $\mathcal{D}$-refresh
$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
$S \leftarrow \mathsf{refresh}(S, I)$
$\boxed{\texttt{IF } \gamma > \gamma_{\max}, \texttt{ THEN } \gamma \leftarrow \gamma_{\max}}$
$c \leftarrow c + \gamma$
$\boxed{T \leftarrow T + 1}$
IF $c \geq \gamma^*$,
  ~~corrupt ← false~~
  $\boxed{\begin{array}{l}\texttt{IF } T^* = 0,\\ \quad T^* \leftarrow T\\ \texttt{IF } T \geq \beta \cdot T^*,\\ \quad \mathsf{corrupt} \leftarrow \mathsf{false}\end{array}}$
OUTPUT (γ, z)

proc. next-ror
$(S, R_0) \leftarrow \mathsf{next}(S)$
$R_1 \xleftarrow{\$} \{0,1\}^\ell$
IF corrupt = true,
  ~~$c \leftarrow 0$~~
  RETURN R_0
ELSE OUTPUT R_b

proc. get-next
$(S, R) \leftarrow \mathsf{next}(S)$
~~IF corrupt = true,~~
  ~~$c \leftarrow 0$~~
OUTPUT R

proc. get-state
$c \leftarrow 0$; corrupt ← true
$\boxed{T \leftarrow 0;\ T^* \leftarrow 0}$
OUTPUT S

proc. set-state(S^*)
$c \leftarrow 0$; corrupt ← true
$\boxed{T \leftarrow 0;\ T^* \leftarrow 0}$
$S \leftarrow S^*$

Fig. 3: Procedures in $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ for $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ with differences from $\mathsf{ROB}(\gamma^*)$ highlighted

Definition 3 (Security of RNG with input against premature next). *A pseudorandom number generator with input* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *is called* $((t, q_{\mathcal{D}}, q_R, q_S), \gamma^*, \gamma_{\max}, \varepsilon, \beta)$-premature-next robust *if for any attacker* $\mathcal{A}$ *running in time at most* t*, making at most* $q_{\mathcal{D}}$ *calls to* $\mathcal{D}$-refresh*,* q_R *calls to* next-ror/get-next *and* q_S *calls to* get-state/set-state*, and any* legitimate *distribution sampler* $\mathcal{D}$ *inside the* $\mathcal{D}$-refresh *procedure, the advantage of* $\mathcal{A}$ *in game* $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ *is at most* ε*.*

Relaxed Security Notions. We note that the above security definition is quite strong. In particular, the attacker has the ability to arbitrarily set the state of $\mathcal{G}$ many times. Motivated by this, we present several relaxed security definitions that may better capture real-world security. These definitions will be useful for our proofs, and we show in Section 4.2 that we can achieve better results for these weaker notions of security:

- $\mathsf{NROB}_{\mathsf{reset}}(\gamma^*, \gamma_{\max}, \beta)$ is $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ in which oracle calls to set-state are replaced by calls to reset-state. reset-state takes no input and simply sets the state of $\mathcal{G}$ to some fixed state S_0, determined by the scheme and sets the entropy counter to zero.[3]
- $\mathsf{NROB}_{\mathsf{1set}}(\gamma^*, \gamma_{\max}, \beta)$ is $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ in which the attacker may only make one set-state call at the beginning of the game.
- $\mathsf{NROB}_{\mathsf{0set}}(\gamma^*, \gamma_{\max}, \beta)$ is $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ in which the attacker may not make any set-state calls.

We define the corresponding security notions in the natural way (See Definition 3), and we call them respectively *robustness against resets*, *robustness against one* set-state, and *robust without* set-state.

4 The Generalized Fortuna Construction

At first, it might seem hopeless to build an RNG with input that can recover from compromise in the presence of premature next calls, since output from a compromised RNG can of course reveal information

[3] Intuitively, this game captures security against an attacker that can cause a machine to reboot.

about the (low-entropy) state. Surprisingly, Ferguson and Schneier presented an elegant away to get around this issue in their Fortuna construction [9]. Their idea is to have several "pools of entropy" and a special "register" to handle next calls. As input that potentially has some entropy comes into the RNG, any entropy "gets accumulated" into one pool at a time in some predetermined sequence. Additionally, some of the pools may be used to update the register. Intuitively, by keeping some of the entropy away from the register for prolonged periods of time, we hope to allow one pool to accumulate enough entropy to guarantee security, even if the adversary makes arbitrarily many premature next calls (and therefore potentially learns the entire state of the register). The hope is to schedule the various updates in a clever way such that this is guaranteed to happen, and in particular Ferguson and Schneier present an informal analysis to suggest that Fortuna realizes this hope in their "constant rate" model (in which the entropy γ_i of each input is an unknown constant).

In this section, we present a generalized version of Fortuna in our model and terminology. In particular, while Fortuna simply uses a cryptographic hash function to accumulate entropy and implicitly assumes perfect entropy accumulation, we will explicitly define each pool as an RNG with input, using the robust construction from [5] (and simply a standard PRG as the register). And, of course, we do not make the constant-rate assumption. We also explicitly model the choice of input and output pools with a new object that we call a scheduler, and we define the corresponding notion of scheduler security. In addition to providing a formal model, we achieve strong improvements over Fortuna's implicit scheduler.

As a result, we prove formally in the standard model that the generalized Fortuna construction is premature-next robust when instantiated with a number of robust RNGs with input, a secure scheduler, and a secure PRG.

4.1 Schedulers

Definition 4. *A* scheduler *is a deterministic algorithm $\mathcal{SC}$ that takes as input a key* skey *and a state $\tau \in \{0,1\}^m$ and outputs a new state $\tau' \in \{0,1\}^m$ and two pool indices,* $\mathsf{in}, \mathsf{out} \in \mathbb{N} \cup \{\bot\}$.

We call a scheduler keyless if there is no key. In this case, we simply omit the key and write $\mathcal{SC}(\tau)$. We say that $\mathcal{SC}$ has P pools if for any skey and any state τ, if $(\tau', \mathsf{in}, \mathsf{out}) = \mathcal{SC}(\mathsf{skey}, \tau)$, then $\mathsf{in}, \mathsf{out} \in [0, P-1] \cup \{\bot\}$.

proc. SGAME
$w_1, \ldots, w_q \leftarrow \mathcal{E}$
$\mathsf{skey} \xleftarrow{\$} \{0,1\}^{|\mathsf{skey}|}$
$\tau_0 \leftarrow \mathcal{A}(\mathsf{skey}, (w_i)_{i=1}^q)$
$(\mathsf{in}_i, \mathsf{out}_i)_{i=1}^q \leftarrow \mathcal{SC}^q(\mathsf{skey}, \tau_0)$
$c \leftarrow 0$; $c_0 \leftarrow 0, \ldots, c_{P-1} \leftarrow 0$; $T^* \leftarrow 0$
FOR T in $1, \ldots, q$,
  IF $w_T > w_{\max}$, THEN OUTPUT 0
  $c \leftarrow c + w_T$; $c_{\mathsf{in}_T} \leftarrow c_{\mathsf{in}_T} + w_T$
  IF $\mathsf{out} \neq \bot$,
    IF $c_{\mathsf{out}_T} \geq 1$, THEN OUTPUT 0
    ELSE $c_{\mathsf{out}_T} \leftarrow 0$
  IF $c \geq \alpha$
    IF $T^* = 0$, THEN $T^* \leftarrow T$
    IF $T \geq \beta \cdot T^*$, THEN OUTPUT 1
OUTPUT 0

Fig. 4: $\mathsf{SGAME}(P, q, w_{\max}, \alpha, \beta)$, the security game for a scheduler $\mathcal{SC}$

Given a scheduler $\mathcal{SC}$ with skey and state τ, we write $\mathcal{SC}^q(\mathsf{skey}, \tau) = (\mathsf{in}_j(\mathcal{SC}, \mathsf{skey}, \tau), \mathsf{out}_j(\mathcal{SC}, \mathsf{skey}, \tau))_{j=1}^q$ to represent the sequence obtained by iteratively computing $(\mathsf{in}, \mathsf{out}, \tau) \leftarrow \mathcal{SC}(\mathsf{skey}, \tau)$ for q times. When $\mathcal{SC}$, skey, and τ are clear or implicit, we will simply write in_j and out_j. We think of in_j as a pool that is to be "filled" at time j and out_j as a pool to be "emptied" immediately afterwards. When $\mathsf{out}_j = \perp$, no pool is emptied.

For a scheduler with P pools, we define the security game $\mathsf{SGAME}(P, q, w_{\max}, \alpha, \beta)$ as in Figure 4. In the security game, there are two adversaries, a sequence sampler $\mathcal{E}$ and an attacker $\mathcal{A}$. We think of the sequence sampler $\mathcal{E}$ as a simplified version of the distribution sampler $\mathcal{D}$ that is only concerned with the entropy estimates $(\gamma_i)_{i=1}^q$. $\mathcal{E}$ simply outputs a sequence of weights $(w_i)_{i=1}^q$ with $0 \le w_i \le w_{\max}$, where we think of the weights as normalized entropies $w_i = \gamma_i/\gamma^*$ and $w_{\max} = \gamma_{\max}/\gamma^*$.

The challenger chooses a key skey at random. Given skey and $(w_i)_{i=1}^q$, $\mathcal{A}$ chooses a start state for the scheduler τ_0, resulting in the sequence $(\mathsf{in}_i, \mathsf{out}_i)_{i=1}^q$. Each pool has an accumulated weight c_j, initially 0, and the pools are filled and emptied in sequence; on the T-th step, the weight of pool in_T is increased by w_T and pool out_T is emptied (its weight set to 0), or no pool is emptied if $\mathsf{out} = \perp$. If at some point in the game a pool whose weight is at least 1 is emptied, the adversary loses. (Remember, 1 here corresponds to γ^*, so this corresponds to the case when the underlying RNG recovers.) We say that such a pool is a *winning pool at time T against* $(\tau_0, (w_i)_{i=1}^q)$. On the other hand, the adversary wins if $\sum_{i=1}^{T^*} w_i \ge \alpha$ and the game reaches the $(\beta \cdot T^*)$-th step (without the challenger winning). Finally, if neither event happens, the adversary loses.

Definition 5 (Scheduler security). *A scheduler $\mathcal{SC}$ with P pools is $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-*secure *if for any pair of adversaries $\mathcal{E}, \mathcal{A}$ with cumulative run-time t, the probability that $\mathcal{E}, \mathcal{A}$ win game $\mathsf{SGAME}(P, q, w_{\max}, \alpha, \beta)$ is at most ε. We call $r = \alpha \cdot \beta$ the* competitive ratio *of $\mathcal{SC}$.*[4]

We note that schedulers are non-trivial objects. Indeed, in Appendix A, we prove the following lower bounds, which imply that schedulers can only achieve superconstant competitive ratios $r = \alpha \cdot \beta$.

Theorem 1. *Suppose that $\mathcal{SC}$ is a $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-secure scheduler running in time $t_{\mathcal{SC}}$. Let $r = \alpha \cdot \beta$ be the competitive ratio. Then, if $q \ge 3$, $\varepsilon < 1/q$, $t = \Omega(q \cdot (t_{\mathcal{SC}} + \log q))$, and $r \le w_{\max}\sqrt{q}$, we have that*

$$r > \log_e q - \log_e(1/w_{\max}) - \log_e \log_e q - 1 \ , \qquad \alpha > \frac{w_{\max}}{w_{\max}+1} \cdot \frac{\log_e(1/\varepsilon) - 1}{\log_e \log_e(1/\varepsilon) + 1} \ .$$

4.2 The Composition Theorem

Our generalized Fortuna construction consists of a scheduler $\mathcal{SC}$ with P pools, P entropy pools $\mathcal{G}_i$, and register ρ. The $\mathcal{G}_i$ are themselves RNGs with input and ρ can be thought of as a much simpler RNG with input which just gets uniformly random samples. On a $\mathsf{refresh}$ call, Fortuna uses $\mathcal{SC}$ to select one pool $\mathcal{G}_{\mathsf{in}}$ to update and one pool $\mathcal{G}_{\mathsf{out}}$ from which to update ρ. next calls are handled entirely from the register.

Formally, we define a generalized Fortuna construction as follows: Let $\mathcal{SC}$ be a scheduler with P pools and let pools $\mathcal{G}_i = (\mathsf{setup}_i, \mathsf{refresh}_i, \mathsf{next}_i)$ for $i = 0, \ldots, P-1$ be RNGs with input. For simplicity, we assume all the RNGs have input length p and output length ℓ, and the same setup procedure, $\mathsf{setup}_i = \mathsf{setup}_{\mathcal{G}}$. We also assume $\mathbf{G} : \{0,1\}^\ell \to \{0,1\}^{2\ell}$ is a pseudorandom generator (without input). We construct a new RNG with input $\mathcal{G}(\mathcal{SC}, (\mathcal{G}_i)_{i=0}^{P-1}, \mathbf{G}) = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ as in Figure 5.

Theorem 2. *Let $\mathcal{G}$ be an RNG with input constructed as above. If the scheduler $\mathcal{SC}$ is a $(t_{\mathcal{SC}}, q_{\mathcal{D}}, w_{\max}, \alpha, \beta, \varepsilon_{\mathcal{SC}})$-secure scheduler with P pools and state length m, the pools $\mathcal{G}_i$ are $((t, q_{\mathcal{D}}, q_R = q_{\mathcal{D}}, q_S), \gamma^*, \varepsilon)$-robust RNGs with input and the register $\mathbf{G}$ is $(t, \varepsilon_{\mathsf{prg}})$-pseudorandom generator, then $\mathcal{G}$ is $((t', q_{\mathcal{D}}, q'_R, q_S), \alpha \cdot \gamma^*, w_{\max} \cdot \gamma^*, \varepsilon', \beta)$-premature-next robust where $t' \approx \min(t, t_{\mathcal{SC}})$ and $\varepsilon' = q_{\mathcal{D}}^2 \cdot q_S \cdot (q_{\mathcal{D}} \cdot \varepsilon_{\mathcal{SC}} + P \cdot 2^m \cdot \varepsilon + q'_R \varepsilon_{\mathsf{prg}})$.*

For our weaker security notions, we achieve better ε':

- *For $\mathsf{NROB}_{\mathsf{reset}}$, $\varepsilon' = q_{\mathcal{D}}^2 \cdot q_S \cdot (q_{\mathcal{D}} \cdot \varepsilon_{\mathcal{SC}} + P \cdot \varepsilon + q'_R \varepsilon_{\mathsf{prg}})$.*
- *For $\mathsf{NROB}_{\mathsf{1set}}$, $\varepsilon' = q_{\mathcal{D}} \cdot \varepsilon_{\mathcal{SC}} + P \cdot 2^m \cdot \varepsilon + q'_R \varepsilon_{\mathsf{prg}}$.*
- *For $\mathsf{NROB}_{\mathsf{0set}}$, $\varepsilon' = q_{\mathcal{D}} \cdot \varepsilon_{\mathcal{SC}} + P \cdot \varepsilon + q'_R \varepsilon_{\mathsf{prg}}$.*

[4] The intuition for the competitive ratio $r = \alpha \cdot \beta$ (which will be explicit in Section 6) comes from the case when the sequence sampler $\mathcal{E}$ is restricted to constant sequences $w_i = w$. In that case, r bounds the ratio between the time taken by $\mathcal{SC}$ to win and the time taken to receive a total weight of one.

proc. setup : $\text{seed}_{\mathcal{G}} \xleftarrow{\$} \text{setup}_{\mathcal{G}}()$ $\text{skey} \xleftarrow{\$} \{0,1\}^{\lvert\text{skey}\rvert}$ OUTPUT seed = $(\text{skey}, \text{seed}_{\mathcal{G}})$	**proc.** refresh(seed, S, I) : PARSE $(\text{skey}, \text{seed}_{\mathcal{G}}) \leftarrow \text{seed}$; $\left(\tau, S_\rho, (S_i)_{i=0}^{P-1}\right) \leftarrow S$ $(\tau, \text{in}, \text{out}) \leftarrow \mathcal{SC}(\text{skey}, \tau)$ $S_{\text{in}} \leftarrow \text{refresh}_{\text{in}}(\text{seed}_{\mathcal{G}}, S_{\text{in}}, I)$ $(S_{\text{out}}, R) \leftarrow \text{next}_{\text{out}}(\text{seed}_{\mathcal{G}}, S_{\text{out}})$ $S_\rho \leftarrow S_\rho \oplus R$ OUTPUT $S = \left(\tau, S_\rho, (S_i)_{i=0}^{P-1}\right)$	**proc.**next(seed, S) : PARSE $\left(\tau, S_\rho, (S_i)_{i=0}^{P-1}\right) \leftarrow S$ $(S_\rho, R) \leftarrow \mathbf{G}(S_\rho)$ OUTPUT $\left(S = \left(\tau, S_\rho, (S_i)_{i=0}^{P-1}\right), R\right)$

Fig. 5: The generalized Fortuna construction

4.3 Proof of Theorem 2

For convenience, we first prove the theorem for the game $\mathsf{NROB}_{\mathsf{1set}}$. (Recall that $\mathsf{NROB}_{\mathsf{1set}}$ is a modified version of NROB in which the adversary is allowed only one call to set-state at the start of the game.) We then show that this implies security for the game NROB and sketch how to extend the proof to the two other notions.

Let us start with some notation to keep track of the state of the security game $\mathsf{NROB}_{\mathsf{1set}}(\alpha \cdot \gamma^*, \beta)$. Most importantly, for each of the P component RNGs $\mathcal{G}_i$ we will keep track of a counter c_i and a corruption flag $\mathsf{corrupt}_i$. These implicitly correspond to the notion of corruption in the basic security game ROB. In particular, the flags are all initially set to $\mathsf{corrupt}_i \leftarrow \mathsf{false}$ and $c_i \leftarrow n$ for each of the RNGs. Whenever the attacker calls $\mathcal{D}$-refresh on our constructed RNG, it receives sample I with min-entropy at least γ, and the scheduler chooses component RNGs $\mathcal{G}_{\mathsf{in}}, \mathcal{G}_{\mathsf{out}}$. Then, we (1) increment $c_{\mathsf{in}} \leftarrow c_{\mathsf{in}} + \gamma$ and if $c_{\mathsf{in}} \geq \gamma^*$ set $\mathsf{corrupt}_{\mathsf{in}} \leftarrow \mathsf{false}$ (2) if $\mathsf{corrupt}_{\mathsf{out}} = \mathsf{true}$ set $c_{\mathsf{out}} = 0$. Whenever the attacker calls set-state or get-state, we set all of the flags $\mathsf{corrupt}_i \leftarrow \mathsf{true}$ and $c_i \leftarrow 0$.

We also define the flag $\mathsf{corrupt}_\rho$ for the register, which is initially set to $\mathsf{corrupt}_\rho \leftarrow \mathsf{false}$. Whenever the attacker calls $\mathcal{D}$-refresh and and the component RNG $\mathcal{G}_{\mathsf{out}}$ selected by the scheduler has $\mathsf{corrupt}_{\mathsf{out}} = \mathsf{false}$ then set $\mathsf{corrupt}_\rho \leftarrow \mathsf{false}$. Whenever the attacker calls set-state, get-state we set $\mathsf{corrupt}_\rho \leftarrow \mathsf{true}$.

We now define a sequence of games:

1. **Game** 0 is the $\mathsf{NROB}_{\mathsf{1set}}(\alpha \cdot \gamma^*, \beta)$ security game against $\mathcal{G}$.
2. **Game** i for $i = 1, \ldots, P$ is a modified version of **Game** 0 in which, whenever we call $\mathsf{next}_{\mathsf{out}}$ at any point in the game on a component RNG $\mathcal{G}_{\mathsf{out}}$ for $\mathsf{out} < i$ and $\mathsf{corrupt}_{\mathsf{out}} = \mathsf{false}$, we choose the output $R \leftarrow \{0,1\}^\ell$ uniformly at random instead of using the output of the RNG.
3. **Game** $P + 1$ is a modified version of **Game** P where, whenever next_ρ is called and $\mathsf{corrupt}_\rho$ is set to false, we output uniform randomness $R \leftarrow \{0,1\}^\ell$.
4. **Game** $P + 2$ is the same as **Game** $P + 1$, but whenever the corrupt flag (the global compromised flag of NROB itself) is set to false we also set $\mathsf{corrupt}_\rho$ to false.

Let $\mathcal{A}$ be an attacker on the $\mathsf{NROB}_{\mathsf{1set}}$ security game running in time t' and making $q_{\mathcal{D}}$ queries to $\mathcal{D}$-refresh, q_R queries to get-next or next-ror, $q_S - 1$ queries to get-state, and at most one set-state query at the very beginning of the game. In each game, we say that $\mathcal{A}$ wins if it guesses the challenge bit $b' = b$.

Claim. For each $i \in \{1, \ldots P\}$ we have $|\Pr[\mathcal{A}$ wins in **Game** $i-1] - \Pr[\mathcal{A}$ wins in **Game** $i]| \leq 2^m\varepsilon$.

Proof. We prove this by reduction to the basic robustness game ROB of the underlying RNG $\mathcal{G}_i$. Assume that there is some distribution sampler $\mathcal{D}$ attacker $\mathcal{A}$ with advantage δ in distinguishing **Game** $i-1$ and **Game** i. The main idea is to compose the distributions sampler $\mathcal{D}$ and the scheduler $\mathcal{SC}$ to create a new distribution sampler $\mathcal{D}'$ that only outputs the samples of $\mathcal{D}$ intended for $\mathcal{G}_i$ and "leaks" all of the other samples to $\mathcal{A}'$. This allows $\mathcal{A}'$ to simulate the $\mathsf{NROB}_{\mathsf{1set}}$ game for $\mathcal{A}$ by knowing the entire state of all the component RNGs except for $\mathcal{G}_i$. The main subtle issue is that the state of the scheduler may get set by the attacker $\mathcal{A}$ in the initial set-state call in a way that depends on the seed of the RNG $\mathcal{G}_i$, preventing $\mathcal{D}'$ from learning the correct sequence of input pools. We handle this by simply guessing the initial scheduler state ahead of time $\tau_{\mathcal{D}'}$. $\mathcal{D}'$ then leaks $\tau_{\mathcal{D}'}$ to $\mathcal{A}'$, and if it happens to be wrong, he just stops the game and outputs a random bit b^*.

In particular, we define a distribution sampler $\mathcal{D}'_{i,q_{\mathcal{D}}}$ (with hard-coded values in the subscript) as shown in Figure 6. We also define $\mathcal{A}'$ as in Figure 7 to essentially simulate the $\mathsf{NROB_{1set}}$ game for $\mathcal{A}$ by using its oracles to get samples for $\mathcal{G}_i$ and knowing the state of all other generators. Let $\tau_{\mathcal{A}}$ be the scheduler state chosen by $\mathcal{A}$ on $\mathsf{set\text{-}state}$ or simply the start state of the scheduler if he does not call $\mathsf{set\text{-}state}$. Let b_{chal} be the challenge bit chosen by the $\mathsf{ROB}(\gamma^*)$ challenger[5] and let b^* be the bit guessed by $\mathcal{A}'$ (which is uniformly random if $\tau_{\mathcal{A}} \neq \tau_{\mathcal{D}'}$). Conditioned on $(b_{chal} = 0) \wedge (\tau_{\mathcal{A}} = \tau_{\mathcal{D}'})$, the view of $\mathcal{A}$ above exactly corresponds to **Game** $i-1$ and conditioned on $(b_{chal} = 1) \wedge (\tau_{\mathcal{A}} = \tau_{\mathcal{D}'})$ it corresponds to **Game** i. Therefore, we have:

$$\begin{aligned}\varepsilon \geq \mathsf{Adv}^{\mathsf{ROB}(\gamma^*)}_{\mathcal{A}',\mathcal{D}'} &= 2 \cdot |\Pr[b^* = b_{chal}] - \frac{1}{2}| \geq |\Pr[b^* = 1 | b_{chal} = 1] - \Pr[b^* = 1 | b_{chal} = 0]| \\ &= \Pr[\tau_{\mathcal{A}} = \tau_{\mathcal{D}'}] |\Pr[b^* = 1 | b_{chal} = 1, \tau_{\mathcal{A}} = \tau_{\mathcal{D}'}] - \Pr[b^* = 1 | b_{chal} = 0, \tau_{\mathcal{A}} = \tau_{\mathcal{D}'}]| \geq 2^{-m}\delta\end{aligned}$$

The second line follows because, conditioned on $\tau_{\mathcal{A}} \neq \tau_{\mathcal{D}'}$, the bit b^* is independent of b_{chal}. This tells us that $\delta \leq 2^m \varepsilon$ as we wanted to show.

proc. $\mathcal{D}'_{i,q_{\mathcal{D}}}(\sigma')$:
IF $\sigma' = 0$ // *initial call*
  $\tau_{\mathcal{D}'} \xleftarrow{\$} \{0,1\}^m$, $\mathsf{skey} \xleftarrow{\$} \{0,1\}^n$, $(\mathsf{in}_j, \mathsf{out}_j)_{j=1}^{q_{\mathcal{D}}} \leftarrow \mathcal{SC}^{q_{\mathcal{D}}}(\mathsf{skey}, \tau_0)$
  $\mathcal{Z}_{sam} \leftarrow \varnothing$, $\mathcal{Z}_{leak} \leftarrow \varnothing$ //*Two empty queues*
  $\sigma \leftarrow 0$
  FOR $j = 1 \ldots q_{\mathcal{D}}$:
    $(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$.
    IF $\mathsf{in}_j = i$, THEN $\mathcal{Z}_{sam}.\mathsf{push}((I, \gamma, z))$
    ELSE $\mathcal{Z}_{leak}.\mathsf{push}((I, \gamma, z))$
  $\sigma' \leftarrow \mathcal{Z}_{sam}$, $I_0 \leftarrow 0$, $\gamma_0 \leftarrow 0$, $z_0 \leftarrow (\mathcal{Z}_{leak}, \tau_{\mathcal{D}'}, \mathsf{skey})$
OUTPUT $(\sigma', I_0, \gamma_0, z_0)$
ELSE
  $\sigma' \leftarrow \mathcal{Z}_{sam}$, $(I, \gamma, z) \leftarrow \mathcal{Z}_{sam}.\mathsf{pop}()$
OUTPUT $(\mathcal{Z}_{sam}, I, \gamma, z)$.

Fig. 6: The distribution sampler $\mathcal{D}'$

□

Next we show that **Game** P is indistinguishable from **Game** $P+1$.

Claim. We have $|\Pr[\mathcal{A}$ wins in **Game** $P] - \Pr[\mathcal{A}$ wins in **Game** $P+1]| \leq 2\varepsilon_{prg}$.

Proof. We prove this by reduction to the PRG security of the underlying "register" $\mathbf{G}$. We simply employ a hybrid argument over all calls to this $\mathbf{G}$ when $\mathsf{corrupt}_\rho = \mathsf{false}$, starting from the earliest, and change the output $(S_\rho, R) \leftarrow \mathbf{G}(S_\rho)$ to being a uniformly random 2ℓ bit value. In each hybrid i the state S_ρ prior to the ith call is either (I) the initial value chosen uniformly random, (II) an output of a prior $\mathbf{G}$ call and therefore uniformly random in this hybrid, (III) some value xored with the output of some pool $\mathcal{G}_i$ when $\mathsf{corrupt}_i$ was set to false and therefore uniformly random. □

Next we show that **Game** $P+1$ is indistinguishable from **Game** $P+2$.

Claim. We have $|\Pr[\mathcal{A}$ wins in **Game** $P+1] - \Pr[\mathcal{A}$ wins in **Game** $P+2]| \leq q_{\mathcal{D}}\varepsilon_{\mathcal{SC}}$.

[5] This does not correspond to the bit b chosen by $\mathcal{A}'$ in the simulation.

proc. $\mathcal{D}$-refresh
$(\tau, \mathsf{in}, \mathsf{out}) \leftarrow \mathcal{SC}(\mathsf{skey}, \tau)$
IF $\mathsf{in} = i$,
  $(\gamma, z) \leftarrow \mathsf{ROB}(\gamma^*).\mathcal{D}\text{-}\mathsf{refresh}()$
ELSE ,
  $(I, \gamma, z) \leftarrow \mathcal{Z}.\mathsf{pop}()$
  $S_{\mathsf{in}} \leftarrow \mathsf{refresh}_{\mathsf{in}}(\mathsf{seed}_{\mathcal{G}}, S_{\mathsf{in}}, I)$
$c_{\mathsf{in}} \leftarrow c_{\mathsf{in}} + \gamma, c \leftarrow c + \gamma$
IF $c_{\mathsf{in}} > \gamma^*$
  $\mathsf{corrupt}_{\mathsf{in}} \leftarrow \mathsf{false}$.
IF $\mathsf{out} = i$,
  $R \xleftarrow{\$} \mathsf{ROB}(\gamma^*).\mathsf{next\text{-}ror}()$
ELSE ,
  $(S_{\mathsf{out}}, R) \leftarrow \mathsf{next}_{\mathsf{out}}(\mathsf{seed}_{\mathcal{G}}, S_{\mathsf{out}})$
  IF $\mathsf{corrupt}_{\mathsf{out}} = \mathsf{true}$
    $c_{\mathsf{out}} \leftarrow 0$
  IF $\mathsf{out} < i$ AND $\mathsf{corrupt}_{\mathsf{out}} = \mathsf{false}$,
    $R \xleftarrow{\$} \{0,1\}^\ell$
$S_\rho \leftarrow S_\rho \oplus R$
OUTPUT (γ, z)

proc. initialize()
$b \xleftarrow{\$} \{0,1\}$
$\mathsf{seed}_{\mathcal{G}} \leftarrow \mathsf{ROB}(\gamma^*).\mathsf{initialize}()$
$(\mathcal{Z}, \tau_{\mathcal{D}'}, \mathsf{skey}) \leftarrow \mathsf{ROB}(\gamma^*).\mathcal{D}\text{-}\mathsf{refresh}()$
$\tau_{\mathcal{A}} \xleftarrow{\$} \{0,1\}^m$
$\tau \leftarrow \tau_{\mathcal{A}}$
FOR $j \in \{0, \dots, P-1\} \setminus \{i\}$:
  $S_j \xleftarrow{\$} \{0,1\}^n$
FOR $j \in \{0, \dots, P-1\}$:
  $c_j \leftarrow n, \mathsf{corrupt}_j \leftarrow \mathsf{false}$
$c \leftarrow n, \mathsf{corrupt} \leftarrow \mathsf{false}$
OUTPUT $\mathsf{seed} = (\mathsf{seed}_{\mathcal{G}}, \mathsf{skey})$

proc. finalize(b^*)
IF $\tau_{\mathcal{D}'} \neq \tau_{\mathcal{A}}$, THEN $b^* \xleftarrow{\$} \{0,1\}$
OUTPUT $\mathsf{ROB}(\gamma^*).\mathsf{finalize}(b^*)$

proc. next-ror
$(S_\rho, R_0) \leftarrow \mathbf{G}(S_\rho)$
$R_1 \xleftarrow{\$} \{0,1\}^\ell$
IF $\mathsf{corrupt} = \mathsf{true}$,
  RETURN R_0
ELSE OUTPUT R_b

proc. get-next
$(S_\rho, R) \leftarrow \mathbf{G}(S_\rho)$
OUTPUT R

proc. get-state
$\mathsf{corrupt} \leftarrow \mathsf{true}$, $c \leftarrow 0$
FOR j in $0, \dots, P-1$
  $c_j \leftarrow 0$, $\mathsf{corrupt}_j \leftarrow \mathsf{true}$
$S_i \leftarrow \mathsf{ROB}(\gamma^*).\mathsf{get\text{-}state}()$
$S \leftarrow (\tau, S_\rho, (S_j)_{j=0}^{P-1})$
OUTPUT S

proc. set-state(S')
$\mathsf{corrupt} \leftarrow \mathsf{true}$, $c \leftarrow 0$
PARSE $(\tau_{\mathcal{A}}, S'_\rho, (S'_j)_{j=0}^{P-1}) \leftarrow S'$
FOR j in $0, \dots, P-1$
  $c_j \leftarrow 0$, $\mathsf{corrupt}_j \leftarrow \mathsf{true}$
  IF $j \neq i$
    $S_j \leftarrow S'_j$
  ELSE
    $\mathsf{ROB}(\gamma^*).\mathsf{set\text{-}state}(S'_j)$
$\tau \leftarrow \tau_{\mathcal{A}}$
$S_\rho \leftarrow S'_\rho$

Fig. 7: Responses of $\mathcal{A}'$ to oracle queries from $\mathcal{A}$

Proof. We prove this by reduction to scheduler security. In particular, Game $P+1$ and $P+2$ can only differ if in Game $P+1$ it happens at some point that the corrupt flag is set to false but $\mathsf{corrupt}_\rho = \mathsf{true}$. We call this event E_{bad}. Intuitively, this corresponds to the case where the attacker makes a get-state or set-state query (causing corrupt and $\mathsf{corrupt}_\rho$ to both be set to true) then sufficient entropy ($\alpha\gamma^*$) has been added by the entropy sampler and sufficient time (βT^*) passes to ensure that corrupt is set to false, but none of the component RNGs $\mathcal{G}_i$ managed to get enough entropy to set $\mathsf{corrupt}_i$ to false or they were never emptied. This corresponds to a failure of the scheduler, and we show how to convert an attacker $\mathcal{A}$ and distribution $\mathcal{D}$ for which $\Pr[E_{bad}] \geq \delta$ into an attack $\mathcal{E}, \mathcal{A}'$ on the scheduler. For convenience, when E_{bad} occurs, let i^* be the index of the first entropy sample given after the last get-state, set-state (compromise) query before E_{bad} occurs.

The attackers $\mathcal{E}, \mathcal{A}'$ guess a random value $i \in [q_{\mathcal{D}}]$ which intuitively corresponds to a guess on i^*. $\mathcal{E}$ simply runs $\mathcal{D}$ for $q_{\mathcal{D}}$ steps to get (among other outputs) the entropy estimates $\{\gamma_j\}$. It outputs the sequence $w_1 = \gamma_i/\gamma^*, w_2 = \gamma_{i+1}/\gamma^*, \dots$. The attacker $\mathcal{A}'(\mathsf{skey})$ simply runs a copy of $\mathcal{A}, \mathcal{D}$ completely simulating **Game** $P+1$ and outputs the state of the scheduler τ immediately before the ith $\mathcal{D}$-refresh query. It is easy to check that $\mathcal{E}, \mathcal{A}'$ win against the scheduler as long as $\mathcal{D}, \mathcal{A}$ cause the event E_{bad} to happen *and* the guess $i = i^*$ is correct. In particular, the entropy counters c_i measuring the amount of entropy added to each RNG behave the same those in the scheduler security game, up to the scaling factor of γ^*. Therefore, they have advantage $\delta/q_{\mathcal{D}}$ which proves the claim. □

Claim. We have $\Pr[\mathcal{A}$ wins in **Game** $P+2] = \frac{1}{2}$.

Proof. The attacker's view in Game $P+2$ is completely independent of challenge bit b. In particular, the next-ror queries with $\mathsf{corrupt} = \mathsf{false}$ always return a random value no matter what the bit b is. Therefore, the attacker's probability of guessing the challenge bit is exactly $\frac{1}{2}$. □

Combining the above claims, we prove the theorem for the case of $\mathsf{NROB}_{\mathsf{1set}}$ security. Notice that the same proof for the game $\mathsf{NROB}_{\mathsf{0set}}$ would not require us to guess the initial state of the scheduler in going from **Game** $i-1$ to **Game** i and would therefore avoid the 2^m factor loss in security.

We can now generically go from $\mathsf{NROB}_{\mathsf{1set}}$ security to full NROB security. Indeed, an analogous version of the same claim can also be used to go from $\mathsf{NROB}_{\mathsf{0set}}$ to $\mathsf{NROB}_{\mathsf{reset}}$ security with the same loss of parameters.

Claim. If an RNG satisfies $(t, q_{\mathcal{D}}, q_R, q_S, \gamma^*, \gamma_{\max}, \varepsilon, \beta)$-$\mathsf{NROB}_{\mathsf{1set}}$ security, then it also satisfies $(t', q_{\mathcal{D}}, q_R, q_S, \gamma^*, \gamma_{\max}, \varepsilon', \beta)$-$\mathsf{NROB}$ security where $t' \approx t, \varepsilon' = q_{\mathcal{D}}^2 q_S \varepsilon$.

Proof. Let $\mathcal{A}$, $\mathcal{D}$ be any attacker and distribution sampler against NROB with advantage δ. Let us divide up the game into at most q_S epochs, where each epoch i begins either at the beginning of the game or with a $\mathsf{set\text{-}state}$ query. Let **Game** 0 be the original NROB game with challenge bit $b = 0$, and let **Game** i be the game where each $\mathsf{next\text{-}ror}$ query in epoch i with $\mathsf{corrupt} = \mathsf{false}$ returns a uniformly random $R \leftarrow \{0,1\}^\ell$. The output of the game is the output of $\mathcal{A}$. We have $|\Pr[(\textbf{Game } 0) \Rightarrow 1] - \Pr[(\textbf{Game } q_S) \Rightarrow 1]| = \delta/2$ meaning that there is some i such that $|\Pr[(\textbf{Game } i) \Rightarrow 1] - \Pr[(\textbf{Game } i+1) \Rightarrow 1]| \geq \delta/(2q_S)$.

We construct $\mathcal{A}', \mathcal{D}'$ with advantage $\delta/(q_S q_{\mathcal{D}}^2)$ in the game $\mathsf{NROB}_{\mathsf{1set}}$. In particular we guess two additional indices $j_{start} < j_{end} \in [q_{\mathcal{D}}]$ for the samples used at the beginning and end of epoch i. The distributions sampler $\mathcal{D}'$ runs $\mathcal{D}$ for $q_{\mathcal{D}}$ times to get all the samples up front, immediately leaks the samples (I_j, γ_j, z_j) for $j < j_{start}$ and $j > j_{end}$, and on each invocation outputs the samples (I_j, γ_j, z_j) starting from $j = j_{start}$ and incrementing j. The attacker $\mathcal{A}'$ simply uses the leaked samples to completely simulate Game i for $\mathcal{A}$ up until the ith epoch. At that point $\mathcal{A}'$ invokes its own challenger for $\mathsf{NROB}_{\mathsf{1set}}$ with distribution sampler $\mathcal{D}'$ and uses the state given by attacker $\mathcal{A}$ in epoch i to make its own $\mathsf{set\text{-}state}$ query. It then uses its oracles to simulate the ith epoch for $\mathcal{A}$. Finally, at the end of the ith epoch $\mathcal{A}'$ again uses the leaked samples to simulate the rest of the game for $\mathcal{A}$. If $\mathcal{A}'$ didn't guess j_{start}, j_{end} correctly, it outputs a random bit. Otherwise it outputs what $\mathcal{A}$ outputs. It's easy to see that if $\mathcal{A}'$ guesses correctly and the challenge bit is $b = 0$ then the above perfectly simulates (**Game** i) and if the bit is $b = 1$ is perfectly simulates (**Game** $i + 1$). Therefore, the advantage of $\mathcal{A}', \mathcal{D}'$ in guessing the challenge bit is $\delta/(q_S q_{\mathcal{D}}^2)$, which proves the claim.

□

5 Instantiating the Construction

5.1 A Robust RNG with Input

Recall that our construction of a premature-next robust RNG with input still requires a robust RNG with input. We therefore present [5]'s construction of such an RNG.

Let $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ be a (deterministic) pseudorandom generator where $m < n$. Let $[y]_1^m$ denote the first m bits of $y \in \{0,1\}^n$. The [5] construction of an RNG with input has parameters n (state length), ℓ (output length), and $p = n$ (sample length), and is defined as follows:

- $\mathsf{setup}()$: Output $\mathsf{seed} = (X, X') \leftarrow \{0,1\}^{2n}$.
- $S' = \mathsf{refresh}(S, I)$: Given $\mathsf{seed} = (X, X')$, current state $S \in \{0,1\}^n$, and a sample $I \in \{0,1\}^n$, output: $S' := S \cdot X + I$, where all operations are over $\mathbb{F}_{2^n}$.
- $(S', R) = \mathsf{next}(S)$: Given $\mathsf{seed} = (X, X')$ and a state $S \in \{0,1\}^n$, first compute $U = [X' \cdot S]_1^m$. Then output $(S', R) = \mathbf{G}(U)$.

Theorem 3 ([5, Theorem 2]). *Let $n > m, \ell, \gamma^*$ be integers and $\varepsilon_{\mathsf{ext}} \in (0,1)$ such that $\gamma^* \geq m + 2\log(1/\varepsilon_{\mathsf{ext}}) + 1$ and $n \geq m + 2\log(1/\varepsilon_{\mathsf{ext}}) + \log(q_{\mathcal{D}}) + 1$. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a deterministic $(t, \varepsilon_{\mathsf{prg}})$-pseudorandom generator. Let $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be defined as above. Then $\mathcal{G}$ is a $((t', q_{\mathcal{D}}, q_R, q_S), \gamma^*, \varepsilon)$-robust RNG with input where $t' \approx t$, $\varepsilon = q_R(2\varepsilon_{\mathsf{prg}} + q_{\mathcal{D}}^2 \varepsilon_{\mathsf{ext}} + 2^{-n+1})$.*

Dodis et al. recommend using AES in counter mode to instantiate their PRG, and they provide a detailed analysis of its security with this instantiation. (See [5, Section 6.1].) We notice that our construction only makes next calls to their RNG during our $\mathsf{refresh}$ calls, and Ferguson and Schneier recommend limiting the number of $\mathsf{refresh}$ calls by simply allowing a maximum of ten per second [9]. They therefore argue that it is reasonable to set $q_{\mathcal{D}} = 2^{32}$ for most security cases (effectively setting a time limit of over thirteen years). So, we can plug in $q_{\mathcal{D}} = q_R = q_S = 2^{32}$.

With this setting in mind, guidelines of [5, Section 6.1] show that our construction can provide a pseudorandom 128-bit string after receiving γ_0^* bits of entropy with γ_0^* in the range of 350 to 500, depending on the desired level of security.

5.2 Scheduler Construction

$$\begin{array}{|l|}\hline
\textbf{proc. } \mathcal{SC}(\mathsf{skey}, \tau): \\
\texttt{IF } \tau \neq 0 \mod P/w_{\max}, \texttt{ THEN } \mathsf{out} \leftarrow \bot \\
\texttt{ELSE } \mathsf{out} \leftarrow \max\{\mathsf{out} : \tau = 0 \mod 2^{\mathsf{out}} \cdot P/w_{\max}\} \\
\mathsf{in} \leftarrow \mathbf{F}(\mathsf{skey}, \tau) \\
\tau' \leftarrow \tau + 1 \mod q \\
\texttt{OUTPUT } (\tau', \mathsf{in}, \mathsf{out}) \\ \hline
\end{array}$$

Fig. 8: Our scheduler construction

To apply Theorem 2, we still need a secure scheduler (as defined in Section 4.1). Our scheduler will be largely derived from Ferguson and Schneier's Fortuna construction [9], but improved and adapted to our model and syntax. In our terminology, Fortuna's scheduler $\mathcal{SC}_{\mathcal{F}}$ is keyless with $\log_2 q$ pools, and its state is a counter τ. The pools are filled in a "round-robin" fashion, (e.g., pool i is filled when $\tau = i \mod \log_2 q$). Every $\log_2 q$ steps, Fortuna empties the maximal pool i such that 2^i divides $\tau / \log_2 q$.

$\mathcal{SC}_{\mathcal{F}}$ is designed to be secure against some unknown but *constant* sequence of weights $w_i = w$. Roughly, if $w > 1/2^i$, then Fortuna will win by the second time that pool i is emptied.[6] We modify Fortuna's scheduler so that it is secure against arbitrary (e.g., not constant) sequence samplers by replacing the round-robin method of filling pools with a pseudorandom sequence. We also slightly lower the number of pools, and we account for $w_{\max}$, as we explain below.

Assume for simplicity that $\log_2 \log_2 q$ and $\log_2(1/w_{\max})$ are integers. We let $P = \log_2 q - \log_2 \log_2 q - \log_2(1/w_{\max})$. We denote by skey the key for some pseudorandom function $\mathbf{F}$ whose range is $\{0, \ldots, P-1\}$. Given a state $\tau \in \{0, \ldots, q-1\}$ and a key skey, we define $\mathcal{SC}(\mathsf{skey}, \tau)$ formally in Figure 8. In particular, the input pool is chosen pseudorandomly such that $\mathsf{in} = \mathbf{F}(\mathsf{skey}, \tau)$. When $\tau = 0 \mod P/w_{\max}$, the output pool is chosen such that out is maximal with $2^{\mathsf{out}} \cdot P/w_{\max}$ divides τ. (Otherwise, there is no output pool.)

Theorem 4. *If the pseudorandom function $\mathbf{F}$ is $(t, q, \varepsilon_{\mathbf{F}})$-secure, then for any $\varepsilon \in (0,1)$, the scheduler $\mathcal{SC}$ defined above is $(t', q, w_{\max}, \alpha, \beta, \varepsilon_{\mathcal{SC}})$-secure with $t' \approx t$, $\varepsilon_{\mathcal{SC}} = q \cdot (\varepsilon_{\mathbf{F}} + \varepsilon)$,*

$$\alpha = 2 \cdot (w_{\max} \cdot \log_e(1/\varepsilon) + 1) \cdot (\log_2 q - \log_2 \log_2 q - \log_2(1/w_{\max})) \ , \qquad \textit{and} \qquad \beta = 4 \ .$$

Remark. *Note that we set $P = \log_2 q - \log_2 \log_2 q - \log_2(1/w_{\max})$ for the sake of optimization. In practice, $w_{\max} = \gamma_{\max}/\gamma^*$ may be unknown, in which case we can safely use $\log_2 q - \log_2 \log_2 q$ pools at a very small cost. We can then still obtaining significant savings in α when $w_{\max} = \gamma_{\max}/\gamma^*$ is small* even if $w_{\max}$ is unknown. *In other words, one can safely instantiate our scheduler (and the corresponding RNG with input) without a bound on $w_{\max}$, and still benefit if $w_{\max}$ happens to be low in practice.*

To prove the theorem, we define a sequence of games. Let **Game** 0 be $\mathsf{SGAME}(P, q, w_{\max}, \alpha, \beta)$ against $\mathcal{SC}$. Let **Game** 1 be **Game** 0 in which the adversary $\mathcal{A}$ is removed and the start state τ_0 is simply selected randomly $\tau_0 \xleftarrow{\$} \{0, \ldots, q-1\}$. Let **Game** 2 be **Game** 1 with $\mathbf{F}(\mathsf{skey}, \cdot)$ replaced by H, a uniformly random function.

Claim. For any sequence sampler $\mathcal{E}$ and any adversary $\mathcal{A}$,

$$\Pr[\mathcal{A}, \mathcal{E} \text{ win in } \textbf{Game } 0] \leq q \cdot \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1]$$

[6] We analyze their construction against constant sequences much more carefully in Section 6.

Proof. Fix $\mathcal{A}, \mathcal{E}$. Let $\tau_0^R \xleftarrow{\$} \{0, \ldots, q-1\}$, and let E be the event that $\mathcal{A}(\mathsf{skey}) = \tau_0^R$. Then,

$$\Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1] \geq \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1|E] \cdot \Pr[E] = \frac{1}{q} \cdot \Pr[\mathcal{A}, \mathcal{E} \text{ win in } \textbf{Game } 0] . \qquad \square$$

Claim. Suppose $\mathbf{F}$ is a $(t, q, \varepsilon_{\mathbf{F}})$-secure pseudorandom function. Then, for any sequence sampler, $\mathcal{E}$ running in time $t' \approx t$,

$$\Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1] \leq \varepsilon_{\mathbf{F}} + \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 2] .$$

Proof. Fix $\mathcal{E}$. We will construct an adversary $\mathcal{A}_{\mathbf{F}}$ that attempts to distinguish between $\mathbf{F}$ under a random key and a uniformly random function.

$\mathcal{A}_{\mathbf{F}}$ receives access to a function H, which is either $\mathbf{F}$ under a random key or uniformly random. $\mathcal{A}_{\mathbf{F}}$ then simulates $\mathcal{E}$, receiving output $(w_1, \ldots, w_q)$. Finally, $\mathcal{A}_{\mathbf{F}}$ simply simulates **Game** 1 with (w_i) and outputs the result of the game.

Note that $\mathcal{A}_{\mathbf{F}}$ outputs exactly the result of $(\textbf{Game } 1)^{\mathcal{E}}$ if H is $\mathbf{F}$ under a random key and exactly the results of $(\textbf{Game } 2)^{\mathcal{E}}$ when H is a random function. The advantage of $\mathcal{A}_{\mathbf{F}}$ in the PRF game is therefore

$$\Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1] + \Pr[\mathcal{E} \text{ loses in } \textbf{Game } 2] - 1 = \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1] - \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 2] .$$

The result follows from the security of $\mathbf{F}$. $\square$

Claim. For any $\varepsilon \in (0, 1)$, let **Game** 2 as above with $\beta = 4$, $P = \log_2 q$, $1/w_{\max}$ an integer, and

$$\alpha = 2 \cdot (w_{\max} \cdot \log_e(1/\varepsilon) + 1) \cdot (\log_2 q - \log_2 \log_2 q - \log_2(1/w_{\max})) .$$

Then, for any sequence sampler $\mathcal{E}$, $\Pr[\mathcal{E} \text{ wins in } \textbf{Game } 2] \leq \varepsilon$.

Proof. Fix the output of $\mathcal{E}$, $(w_1, \ldots, w_q)$. Let $\tau_0 \in \{0, \ldots, q-1\}$ be some start state with the corresponding sequence $(\mathsf{in}_i, \mathsf{out}_i)_{i=1}^q$. Note that in_i is uniformly random and independent of $\mathcal{E}, \tau_0$.

Let T^* such that $\sum_{i=1}^{T^*} w_i \geq \alpha$. Let j such that $2^j \geq w_{\max} \cdot T^*/P > 2^{j-1}$. (If no such T^*, j exist, then $\mathcal{SC}$ wins by default.)

We wish to find a pool that was not emptied before time T^* but is emptied relatively soon after time T^*. Call the first such pool to be emptied win and the first time that pool win is emptied T_{win}. Note that there is at most one $k \geq j$ such that pool k was emptied before time T^*. If such a pool exists, call the first time that it is emptied T_k. Note that $2^j \cdot P/w_{\max}$ divides $T_k + \tau_0$. We consider three different cases:

1. If no such k exists, then some pool whose index is at least j must be emptied by $2^j \cdot P_b/w_{\max}$, and by hypothesis it cannot have been emptied before time T^*. So $T_{\mathsf{win}} \leq 2^j \cdot P/w_{\max}$.
2. If $k > j$, then pool k is emptied at most every $2^{j+1} \cdot P/w_{\max}$ rounds, so the pool emptied at time $T_k + 2^j \cdot P/w_{\max}$ cannot be pool k. But, $2^j \cdot P/w_{\max}$ divides $T_k + 2^j \cdot P/w_{\max} + \tau_0$, so some pool whose index is at least j must be emptied at time $T_k + 2^j \cdot P/w_{\max}$. Therefore, $T_{\mathsf{win}} = T_k + 2^j \cdot P/w_{\max} < T^* + 2^j \cdot P/w_{\max}$.
3. If $k = j$, then $2^{j+1} \cdot P/w_{\max}$ does not divide $T_k + \tau_0$, and therefore $2^{j+1} \cdot P/w_{\max}$ must divide $T_k + 2^j \cdot P/w_{\max}$. So, a pool whose index is greater than j must be emptied at that time. Therefore $T_{\mathsf{win}} \leq T_k + 2^j \cdot P/w_{\max} < T^* + 2^j \cdot P/w_{\max}$.

   In all cases,

$$T_{\mathsf{win}} < T^* + 2^j \cdot P/w_{\max} \leq 2^{j+1} \cdot P/w_{\max} .$$

So $T_{\mathsf{win}} < \frac{2^{j+1}}{2^{j-1}} \cdot T^* = 4 \cdot T^* = \beta \cdot T^*$. Recall that the scheduler wins if it empties a pool with weight at least one at any time before $\beta \cdot T^*$. Therefore, the scheduler wins if win has weight at least one after time T^*.

Let $0 \leq W_{\mathsf{win},i} \leq w_{\max}$ be the random variable that takes value w_i if $\mathsf{in}_i = \mathsf{win}$ and 0 otherwise. Then, the weight of pool win at time T^* is $\sum_{i=1}^{T^*} W_{\mathsf{win},i}$.

We recall the standard Chernoff-Hoeffding bound:

$$\Pr[W \leq (1-\delta)\mu] \leq e^{-\delta^2 \, \mathbb{E}[W]/(2w_{\max})}$$

for any $\delta \in (0,1)$. Plugging in, the probability that the scheduler loses after starting in state τ_0 is at most

$$\Pr_H\Big[\sum_{i\le T^*} W_{\mathsf{win},i} \le 1\Big] \le e^{-\frac{\alpha_b}{2w_{\max}\cdot P}\cdot(1-\frac{2P}{\alpha})} = e^{1/w_{\max}} \cdot e^{-\frac{\alpha}{2w_{\max}\cdot P}} .$$

Finally, we set $\varepsilon = e^{1/w_{\max}} \cdot e^{-\frac{\alpha}{2w_{\max}\cdot P}}$ and solve for α:

$$\begin{aligned}\alpha &= 2\cdot(w_{\max}\cdot \log_e(1/\varepsilon)+1)\cdot P\\ &= 2\cdot(w_{\max}\cdot \log_e(1/\varepsilon)+1)\cdot(\log_2 q - \log_2\log_2 q - \log_2(1/w_{\max})) .\end{aligned}$$ □

Putting everything together, for any $\mathcal{E}, \mathcal{A}$,

$$\begin{aligned}\varepsilon_{\mathcal{SC}} &\le q\cdot \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 1]\\ &\le q\cdot(\varepsilon_{\mathbf{F}} + \Pr[\mathcal{E} \text{ wins in } \textbf{Game } 2])\\ &\le q\cdot(\varepsilon_{\mathbf{F}}+\varepsilon)\end{aligned}$$

5.3 Scheduler Instantiation

To instantiate the scheduler in practice, we suggest using AES as the PRF $\mathbf{F}$. As in [5], we ignore the computational error term $\varepsilon_{\mathbf{F}}$ and set $\varepsilon_{\mathcal{SC}} \approx q\varepsilon$.[7] In our application, our scheduler will be called only on refresh calls to our generalized Fortuna RNG construction, so we again set $q = 2^{32}$. It seems reasonable for most realistic scenarios to set $w_{\max} = \gamma_{\max}/\gamma^* \approx 1/16$ and $\varepsilon_{\mathcal{SC}} \approx 2^{-192}$, but we provide values for other $w_{\max}$ and ε as well:

$\varepsilon_{\mathcal{SC}}$	q	$w_{\max}$	α	β	P
2^{-128}	2^{32}	$1/64$	115	4	21
2^{-128}	2^{32}	$1/16$	367	4	23
2^{-128}	2^{32}	$1/4$	1445	4	25
2^{-128}	2^{32}	1	6080	4	27

$\varepsilon_{\mathcal{SC}}$	q	$w_{\max}$	α	β	P
2^{-192}	2^{32}	$1/64$	144	4	21
2^{-192}	2^{32}	$1/16$	494	4	23
2^{-192}	2^{32}	$1/4$	2000	4	25
2^{-192}	2^{32}	1	8476	4	27

$\varepsilon_{\mathcal{SC}}$	q	$w_{\max}$	α	β	P
2^{-256}	2^{32}	$1/64$	174	4	21
2^{-256}	2^{32}	$1/16$	622	4	23
2^{-256}	2^{32}	$1/4$	2554	4	25
2^{-256}	2^{32}	1	$10,871$	4	27

5.4 Putting It All Together

Now, we have all the pieces to build an RNG with input that is premature-next robust (by Theorem 2). Again setting $q = 2^{32}$ and assuming $w_{\max} = \gamma_{\max}/\gamma^* \approx 32/500 \approx 1/16$, our final scheme can output a secure 128-bit key in four times the amount of time that it takes to receive roughly 20 to 30 kilobytes of entropy.

6 Constant-Rate Adversaries

We note that the numbers that we achieve in Section 5.4 are not ideal. But, our security model is also very strong. So, we follow Ferguson and Schneier [9] and consider the weaker model in which the distribution sampler $\mathcal{D}$ is restricted to a constant entropy rate. While this model may be too restrictive, it leads to interesting results, and it suggests that our construction (or, rather, the slight variant suggested in Section 6.3) may perform much better against distribution samplers that are not too adversarial. Indeed, if we think of the distribution sampler $\mathcal{D}$ as essentially representing nature, this might not be too unreasonable.

Constant-Rate Model. We simply modify our definitions in the natural way. First, we say that a distribution (resp., sequence) sampler is *constant* if, for all i, $\gamma_i = \gamma$ (resp., $w_i = w$) for all i for some fixed γ (resp., w). Second, we say that an RNG with input is $((t, q_{\mathcal{D}}, q_R, q_S), \gamma^*, \gamma_{\max}, \varepsilon, \beta)$*-premature-next robust against constant adversaries* if it is $((t, q_{\mathcal{D}}, q_R, q_S), \gamma^*, \gamma_{\max}, \varepsilon, \beta)$-premature-next robust when the distribution sampler $\mathcal{D}$ is required to be constant. Third, we say that a scheduler is $(t, q, w_{\max}, r, \varepsilon)$*-secure against constant sequences* if, for some[8] α, β such that $\alpha\cdot\beta = r$ it is $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-secure when the

[7] [5] contains a detailed discussion of the subtleties here and the justification for such an assumption.

[8] We note that when the sequence sampler $\mathcal{E}$ must be constant, $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-security is equivalent to $(t, q, w_{\max}, \alpha', \beta', \varepsilon)$-security if $\alpha\cdot\beta = \alpha'\cdot\beta'$.

sequence sampler $\mathcal{E}$ is required to be constant. When $\varepsilon = 0$ and the adversaries are allowed unbounded computation (as is the case in our construction), we simply leave out the parameters t and ε.

Finally, we note that our composition theorem, Theorem 2, applies equally well in the constant-rate case. In particular, replacing a secure scheduler with a scheduler that is secure against constant sequences results in an RNG with input that is premature-next robust against constant adversaries, with identical parameters. This will allow us to achieve much better parameters for schedulers and RNGs with input against constant adversaries.

6.1 Optimizing Fortuna's Scheduler

Ferguson and Schneier essentially analyze the security of a scheduler that is a deterministic version of our scheduler from Section 5.2, with pseudorandom choices replaced by round-robin choices [9]. (This is, of course, where we got the idea for our scheduler.) They conclude that it achieves a competitive ratio of $2\log_2 q$. However, the correct value is $3\log_2 q$.[9] Ferguson and Schneier's model differs from ours in that they do not consider adversarial starting times τ_0 between the emptying of pools. Taking this (important) consideration into account, it turns out that $\mathcal{SC}_{\mathcal{F}}$ achieves a competitive ratio of $r_{\mathcal{F}} = 3.5\log_2 q$ in our model (e.g., for $q = 2^{32}$, we get $r_{\mathcal{F}} = 112$, as opposed to their claimed value of 64).[10]

Interestingly, the pseudocode in [9] actually describes a potentially stronger scheduler than the one that they analyzed. Instead of emptying just pool i, this new scheduler empties *each* pool j with $j \le i$. Although Ferguson and Schneier did not make use of this in their analysis, we observe that this would lead to significantly improved results provided that the scheduler could "get credit" for all the entropy from *multiple* pools. Unfortunately, our model syntactically cannot capture the notion of multiple pools being emptied at once, and this is necessary for our composition theorem (Theorem 2). Fortunately, we notice that our model can simulate a multiple-pool scheduler by simply treating any set of pools that is emptied together at a given time as one new pool.

In Appendix B, we make this observation concrete and further optimize the scheduler of Fortuna to obtain the following result.

Theorem 5. *For any integer $b \ge 2$, there exists a keyless scheduler $\mathcal{SC}_b$ that is $(q, w_{\max}, r_b)$-secure against constant sequences where*

$$r_b = \left(b + \frac{w_{\max}}{b} + \frac{1 - w_{\max}}{b^2}\right) \cdot (\log_b q - \log_b \log_b q - \log_b(1/w_{\max})) \ .$$

In particular, with $w_{\max} = 1$ and $q \to \infty$, $b = 3$ is optimal with $r_3 \approx 2.1\log_2 q \approx \frac{r_{\mathcal{F}}}{1.66} \approx \frac{r_2}{1.19} \approx \frac{r_4}{1.01}$.

We note that $\mathcal{SC}_b$ performs even better in the non-asymptotic case. For example, in the case that Ferguson and Schneier analyzed, $q = 2^{32}$ and $w_{\max} = 1$, we have $r_3 \approx 58.2 \approx \frac{r_{\mathcal{F}}}{1.9}$, saving almost half the entropy compared to Fortuna.

6.2 Constant-Rate Instantiation

Using the results from above, we note that applying our generalized Fortuna construction with the scheduler from Appendix B with $b = 3$, $q = 2^{32}$, and $w_{\max} = 1$ yields an RNG with input that can achieve a secure 128-bit key after accumulating 3 to 4.5 kilobytes of entropy *from a constant distribution sampler $\mathcal{D}$*. So, this constant-rate construction (in this restricted setting) is over twenty-five more efficient than our general construction.[11] (In Section 6.3, we present a scheduler that achieves these better results in the constant-rate case but also achieves the results presented in Section 5 in our stronger model.)

---

[9] There is an attack: Let $w = 1/(2^i + 1)$ and start Fortuna's counter so that pool $i+1$ is emptied after $2^i \cdot \log_2 q$ steps. Clearly, $\mathcal{SC}_{\mathcal{F}}$ takes $(2^i + 2^{i+1}) \cdot \log_2 q = 3 \cdot 2^i \cdot \log_2 q$ total steps to finish, achieving a competitive ratio arbitrarily close to $3\log_2 q$.

[10] This follows from the analysis of our own scheduler in Appendix B.

[11] To compare with our previous numbers from Section 5, recall that we had $\beta = 4$. Therefore, we note that the above scheduler achieves such security in four times the amount of time that it takes to receive about 750 bytes to 1.2 kilobytes of entropy. These are the proper numbers to compare, though they make less sense in the constant-rate case.

Ferguson and Schneier claim in [9] that their underlying seed (the key for AES in counter mode) reaches a secure 128-bit key after receiving what amounts to over 1.7 kilobytes of entropy (after accounting for the error and difference in models mentioned in Section 6). However, we note that they implicitly assume that their construction achieves perfect entropy accumulation. We achieve formally provable security and lose roughly a factor of four from using the construction of [5] described in Section 5 to accumulate entropy, though due to various optimizations we manage to come within a factor of about 2 of Ferguson and Schneier's claim.

6.3 A Scheduler Secure in Both Worlds

Recall that in Section 5.2, we construct a secure scheduler, and above we construct a keyless scheduler that is secure only against constant sequence samplers but achieves much better parameters. We justify this weaker model by arguing that, in practice, a purely adversarial distribution sampler may be too stringent. We would like to say that the "true" security of our construction in a "real world" setting lies somewhere in between. And, we would like to say that practitioners can use one scheduler that is provably secure in the stronger model and achieves excellent parameters when adversaries happen to be friendlier.

However, this is unfortunately not true for the scheduler that we presented in Section 5.2. Recall that this scheduler selected which pool to fill at a given time pseudorandomly, using a PRF. It is not hard to see that its performance against constant sequence samplers is only slightly better than its performance against arbitrary adversaries. Intuitively, our keyless scheduler distributes weight evenly amongst all of its pools, while our more secure scheduler only does so *in expectation*. As a result, it can put entropy in the "wrong pool" with fairly high probability, even in the constant-rate case.

Luckily, there is a fairly simple solution. Instead of selecting a new pool pseudorandomly at each step, we instead choose a pseudorandom permutation of all P pools every P steps. In particular, given a state τ and a key skey, the scheduler computes $\pi \leftarrow \mathbf{F}(\mathsf{skey}, \lfloor \tau/P \rfloor)$ where π is a permutation of P elements, $\mathbf{F}$ is a pseudorandom function whose range is all permutations on P elements, and P is the number of pools of the scheduler. It then fills pool $\mathsf{in} \leftarrow \pi(\tau \mod P)$. The scheduler can otherwise behave like our scheduler from Section 6. It is not hard to see that our proofs of security in both the constant-rate and general case apply immediately to this modified scheduler. So, we recommend that practitioners implement this construction.

7 Relaxing the Seed Independence of the Distribution Sampler

In this section, we address another limitation of the original model of [5], which our model inherits: the subtle issue of *seed independence*. In particular, the model of [5] does not allow the distribution sampler $\mathcal{D}$ to have access to the initial seed seed of the RNG with input.

As explained by [5], this is *necessary* to some extent, as there is a very simple impossibility result when $\mathcal{D}$ knows the seed. Given any RNG with input $\mathcal{G}$ whose input length is $p \geq 2$, consider $\mathcal{D}$ that simply samples $I_1, \ldots, I_{q_\mathcal{D}}$ uniformly such that $\mathsf{next}(\mathsf{seed}, S_{q_\mathcal{D}})$ starts with a 0 where $S_0 = 0$, and $S_j = \mathsf{refresh}(\mathsf{seed}, S_{j-1}, I_j)$. Let $\mathcal{A}$ be the adversary that simply calls $\mathsf{set\text{-}state}(0)$, makes $q_\mathcal{D}$ calls to $\mathcal{D}$-$\mathsf{refresh}$, calls $\mathsf{next\text{-}ror}$, and simply outputs the first bit of the resulting output. It is clear that this pair of $\mathcal{A}$ and $\mathcal{D}$ will break the RNG security, and also that $\mathbf{H}_\infty(I_j | I_1, \ldots, I_{j-1}, I_{j+1}, \ldots I_{q_\mathcal{D}}) \approx p - 1$.

In fact, the original provably secure scheme from [5] can be attacked much more dramatically (than the above generic attack) by a seed-dependent $\mathcal{D}$. Recall, in that scheme part of the seed X, input I, and state S are simply elements in a finite field $\mathbb{F}_{2^n}$. Also, if the start state S is 0 and the distribution sampler $\mathcal{D}$ samples some random variables $I_1, \ldots, I_{q_\mathcal{D}}$, then after $q_\mathcal{D}$ $\mathsf{refresh}$ calls the resulting state will be $S = X^{q_\mathcal{D}-1} I_1 + X^{q_\mathcal{D}-2} I_2 + \ldots + I_{q_\mathcal{D}}$. This suggests a natural attack: simply let I_j be sampled uniformly from $\{0, X^{j-q_\mathcal{D}}\}$. Clearly the distribution sampler provides $q_\mathcal{D}$ bits of entropy in this case, but a quick check shows that the state S is the sum of uniformly random bits, so it can be only 0 or 1. The distribution sampler can therefore provide arbitrarily large amounts of entropy while only letting the state accumulate one bit.

Unfortunately, our generalized Fortuna scheme that is premature-next robust suffers a similar fate, even *without* attacking any of the "pool" RNGs. Indeed, if the distribution sampler $\mathcal{D}$ has access to the seed, then in particular, it has access to the key skey of the scheduler. $\mathcal{D}$ can therefore choose to only provide entropy to pools that will soon be emptied. For example, against our scheduler in Section 5.2, $\mathcal{D}$ can provide 1 bit of entropy whenever pool 0 will be filled next, and no entropy otherwise. If the adversary $\mathcal{A}$ then calls get-next repeatedly after every $\mathcal{D}$-refresh call, the RNG will never accumulate any entropy (with high probability).

To sum up, existing schemes *crucially* rely on the seed-independence of the distribution sampler, and it is also clear that full seed-dependence is *impossible*. Finding the right (realistic and, yet, provably secure) balance between these extremes is an important subject for further research. In the next subsection, we make some initial progress along these lines by introducing a somewhat realistic model that effectively allows a certain level of seed dependence.

7.1 Semi-Adaptive set-refresh

Our extended adversarial model is motivated by the following realistic scenario given by Ferguson and Schneier when describing Fortuna [9]. They assume that there are several sources of entropy $N_1, N_2, \ldots$ contributing the inputs I_j for the $\mathcal{D}$-refresh procedure. Some of these sources might be completely controlled by the attacker $\mathcal{A}$, while others are assumed to provide "good" entropy. Of course, since the actual RNG does not know the identity of these adversarial sources, they suggest that the RNG should take the inputs from $N_1, N_2, \ldots$ in a round-robin manner, ensuring that "good" sources periodically contribute fresh entropy to the system.

Semi-Adaptive set-refresh Model. Translating this natural attack scenario to our model (for both ROB and NROB), we can think of the union of "good" sources N_i as our original (seed-independent) distribution sampler $\mathcal{D}$, while the union of "bad" source N_i can be modeled by giving the (seed-dependent) attacker $\mathcal{A}$ access to the simple set-refresh oracle shown in Figure 9.

proc. set-refresh(I^*)
$S \leftarrow$ refresh(S, I^*)

Fig. 9: The set-refresh oracle

Note, in particular, that since set-refresh is called by $\mathcal{A}$, the entropy counter c is not incremented during this call. Additionally, since in the above motivating example the RNG will call the good/bad entropy sources in a round-robin manner, it seems reasonable to make the assumption that the *order* of set-refresh calls is *seed-independent* (though, crucially, the *values* I^* in various set-refresh(I^*) calls *can* depend on the seed).[12] Overall, we can think of $\mathcal{A}$ and $\mathcal{D}$ as defining a partially seed-dependent distribution sampler $\mathcal{D}'$.

We arrive at the following natural extension of robustness, which we call the *semi-adaptive* set-refresh *model*, where $q_{\mathcal{D}}$ is now the maximal sum of the number $\mathcal{D}$-refresh and the set-refresh calls made by $\mathcal{A}$:

- $\mathcal{D}$ selects a subset of indices $J \subseteq \{1, \ldots, q_{\mathcal{D}}\}$ where set-refresh calls will be made.
- $\mathcal{A}$ learns seed and J, and can play the usual ROB/NROB game, except the sequence of its $\mathcal{D}$-refresh and set-refresh calls must be consistent with J. I.e., the j-th such call must be set-refresh iff $j \in J$.

Security Against Semi-adaptive set-refresh. We observe that the robustness proofs of both the original RNG construction of [5] and our generalized Fortuna construction easily extend to handle *semi-adaptive* set-refresh calls. Indeed, we even achieve identical parameters.

Interestingly, we are not aware of an attack on [5]'s construction even with seed-dependent (i.e., fully-adaptive) set-refresh calls, but our current proof crucially uses semi-adaptivity. Unfortunately, our attack on generalized Fortuna with a seed-dependent distribution sampler easily extends to an attack using seed-dependent (i.e., fully-adaptive) set-refresh calls instead. Indeed, using skey, $\mathcal{A}$ can schedule set-refresh calls such that $\mathcal{D}$-refresh calls only affect pools that will soon be emptied.

[12] Note that, while this assumption is quite strong, we do not impose a fixed order on the set-refresh calls or assume constant entropy from $\mathcal{D}$-refresh calls as [9] do. Indeed, the original Fortuna construction is clearly not secure in our extended model even with a constant entropy assumption.

Theorem 6. *The security bound for the RNG of [5] given in Theorem 3 extends to the semi-adaptive* set-refresh *model. Similarly, the premature next robustness of the generalized Fortuna scheme given in Theorem 2 extends to the semi-adaptive* set-refresh *model, provided all the pool RNGs $\mathcal{G}_i$ are robust in the semi-adaptive* set-refresh *model.*

Since both proofs are simple variants of the original proofs, we will only sketch the key steps required to extend both proofs below.

Extending the Composition Theorem. We first show how to extend the proof of our main composition theorem (Theorem 2) to handle semi-adaptive set-refresh. To do so, we need to show how to extend the main reduction, mapping the "big" attackers $\mathcal{A}, \mathcal{D}$ against the composed RNG $\mathcal{G}$ into "small" attackers $\mathcal{A}_i, \mathcal{D}_i$ against the pool RNG $\mathcal{G}_i$, to the semi-adaptive set-refresh setting. Fortunately, this is simple because the scheduler key skey in our reduction is selected directly by $\mathcal{D}_i$ (see Figure 6) and then immediately passed to $\mathcal{A}_i$ via leakage. In particular, $\mathcal{D}_i$ can now also compute the index set J , then use skey to "project" this set J to whatever calls $j \in J$ will be "routed" to $\mathcal{G}_i$ by the scheduler, and finally pass this "projected set" J_i to the challenger. $\mathcal{A}_i$ then learns the seed and J_i and can simulate the run of $\mathcal{A}$ as before (see Figure 7), handling set-refresh calls in the obvious way.

Extending [5]'s Proof. Next, we sketch the changes needed to extend the original proof of robustness of the [5] construction (see Section 5.1) to handle semi-adaptive set-refresh calls. The proof of [5] consists of three steps: (1) reducing robustness to two simpler properties called *preserving* and *recovering* security (see [5]'s Theorem 1); (2) showing preserving security; and (3) showing recovering security. Step (1) easily extends to semi-adaptive set-refresh calls, provided the notion of recovering security is naturally augmented to include semi-adaptive set-refresh calls. Step (2) needs no changing at all (as preserving security already gives $\mathcal{A}$ access to a *fully* adaptive set-refresh oracle). Hence, it suffices to show how to extend the proof of recovering security in step (3) to a slightly modified version that includes set-refresh calls. We present the modified recovery security game together with the preserving security game and a modified version of [5]'s composition theorem in Appendix C.

Intuitively, recovering security considers an attacker that sets the state to some arbitrary value S_0 and starts the distribution sampler $\mathcal{D}$ after k calls to $\mathcal{D}$-refresh. Following that, d calls to $\mathcal{D}$-refresh are made, resulting in final state S, where d, k are chosen by $\mathcal{A}$ such that the corrupt flag is false after the d calls to $\mathcal{D}$-refresh. Then, the attacker $\mathcal{A}$ attempts to distinguish the *full* output $(S^*, R) \leftarrow \mathsf{next}(S)$ from uniform. In our modified version, an index set J is chosen by $\mathcal{D}$ at the beginning, and the j-th $\mathcal{D}$-refresh call is replaced by a set-refresh call if and only if $j \in J$.

Note that in the recovering game, [5]'s RNG with input effectively computes a function of the form

$$h^*_{X,X'}(I_1, \ldots, I_d) = \Big[X' \cdot \sum_{j=0}^{d-1} I_{d-j} \cdot X^j\Big]_1^m + [X' \cdot S_0]_1^m$$

and applies a PRG $\mathbf{G}$ to the result. In [5], the authors show that recovering security follows immediately from the fact that $h^*_{X,X'}$ is a good randomness extractor. In particular, if the sum of the conditional min-entropies of the input is sufficiently high (i.e., above γ^*) and the I_j are chosen independently of X, X', then $(X, X', h^*_{X,X'}(I_1, \ldots, I_D))$ is $\varepsilon_{\mathsf{ext}}$-close to uniform (with $\varepsilon_{\mathsf{ext}}$ defined as in Theorem 3).

Our key observation is simply that $h^*_{X,X'}$ is linear. Intuitively, we wish to define three sequences: $I_j^{\mathcal{D},\mathcal{A}}(\mathsf{seed})$ is the sequence of inputs to refresh calls, including both $\mathcal{D}$-refresh and set-refresh; $I_j^{\mathcal{D}}$ is the contribution from $\mathcal{D}$-refresh calls; and $I_j^{\mathcal{A}}(\mathsf{seed})$ is the contribution from $\mathcal{A}$'s set-refresh calls. We then want to say that $h^*_{X,X'}$ applied to $I_j^{\mathcal{D},\mathcal{A}}$ is the sum of $h^*_{X,X'}$ applied to each adversary's contribution.

In particular, fix $\mathcal{A}, \mathcal{D}$. Let $(I_j)_{j=1}^{q_{\mathcal{D}}}$ be the distributions sampled by $\mathcal{D}$; J the index set chosen by $\mathcal{D}$, $\mathsf{seed} = (X, X')$ a randomly chosen seed; k, d the (seed-dependent) choices of $\mathcal{A}$; and $(I_j^*(\mathsf{seed}))_{j \in J}$ the input of $\mathcal{A}$ to set-refresh calls. Then, formally, we let $I_j^{\mathcal{D}} = I_j^{\mathcal{D},\mathcal{A}}(\mathsf{seed}) = I_j$ and $I_j^{\mathcal{A}}(\mathsf{seed}) = 0$ if $j \notin j$, and

$I_j^{\mathcal{A}}(\mathsf{seed}) = I_j^{\mathcal{D},\mathcal{A}}(\mathsf{seed}) = I_j^*(\mathsf{seed})$ and $I_j^{\mathcal{D}} = 0$ if $j \in J$. We can then write

$$\begin{aligned} U &:= h^*_{X,X'}\big(I^{\mathcal{A},\mathcal{D}}_{k+1}(\mathsf{seed}),\ldots,I^{\mathcal{A},\mathcal{D}}_{k+d}(\mathsf{seed})\big) + [X'\cdot S_0]_1^m \\ &= h^*_{X,X'}\big(I^{\mathcal{D}}_{k+1},\ldots,I^{\mathcal{D}}_{k+d}\big) + h^*_{X,X'}\big(I^{\mathcal{A}}_{k+1}(\mathsf{seed}),\ldots,I^{\mathcal{A}}_{k+d}(\mathsf{seed})\big) + [X'\cdot S_0]_1^m \ . \end{aligned}$$

Finally, we simply note that $I_j^{\mathcal{D}}$ are chosen independently from X, X' (equivalently, they are the output of some valid distribution sampler $\mathcal{D}'$), and therefore the proof of [5] implies that $(X, X', h^*_{X,X'}(I^{\mathcal{D}}_{k+1},\ldots,I^{\mathcal{D}}_{k+d}))$ is $\varepsilon_{\mathsf{ext}}$ close to uniform when the sum of the entropies of the corresponding distributions is sufficiently high. This of course immediately implies that X, X', U is also $\varepsilon_{\mathsf{ext}}$ close to uniform. The result, presented below, then follows immediately from the proof in [5].

Theorem 7. *Let $n > m, \ell, \gamma^*$ be integers and $\varepsilon_{\mathsf{ext}} \in (0,1)$ such that $\gamma^* \geq m + 2\log(1/\varepsilon_{\mathsf{ext}}) + 1$ and $n \geq m+2\log(1/\varepsilon_{\mathsf{ext}})+\log(q_{\mathcal{D}})+1$. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a deterministic $(t, \varepsilon_{\mathsf{prg}})$-pseudorandom generator. Let $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be defined as in Section 5. Then $\mathcal{G}$ is a $((t', q_{\mathcal{D}}, q_R, q_S), \gamma^*, \varepsilon)$-robust RNG with input* in the semi-adaptive set-refresh model *where $t' \approx t$, $\varepsilon = q_R(2\varepsilon_{\mathsf{prg}} + q_{\mathcal{D}}^2\varepsilon_{\mathsf{ext}} + 2^{-n+1})$.*

Combining Theorems 6 and 7, we see that the security of the instantiation that we presented in Section 5 immediately extends to the semi-adaptive set-refresh model with identical parameters.

References

1. Barak, B., and Halevi, S. A model and architecture for pseudo-random generation with applications to /dev/random. pp. 203–212.
2. Barker, E., and Kelsey, J. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A, 2012.
3. Bellare, M., and Rogaway, P. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology - EUROCRYPT 2006*, S. Vaudenay, Ed., vol. 4004 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 409–426.
4. CVE-2008-0166. Common Vulnerabilities and Exposures, 2008.
5. Dodis, Y., Pointcheval, D., Ruhault, S., Vergniaud, D., and Wichs, D. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 647–658.
6. Dorrendorf, L., Gutterman, Z., and Pinkas, B. Cryptanalysis of the random number generator of the windows operating system. *IACR Cryptology ePrint Archive 2007* (2007), 419.
7. Eastlake, D., Schiller, J., and Crocker, S. *RFC 4086 - Randomness Requirements for Security*, June 2005.
8. Ferguson, N. Private communication, 2013.
9. Ferguson, N., and Schneier, B. *Practical Cryptography*, 1 ed. John Wiley & Sons, Inc., New York, NY, USA, 2003.
10. Gutterman, Z., Pinkas, B., and Reinman, T. Analysis of the linux random number generator. pp. 371–385.
11. Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium* (Aug. 2012).
12. Information technology - Security techniques - Random bit generation. ISO/IEC18031:2011, 2011.
13. Kelsey, J., Schneier, B., and Ferguson, N. 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* (1999), Springer, pp. 13–33.
14. Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, S. Vaudenay, Ed., vol. 1372 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 168–188.
15. Killmann, W. and Schindler, W. A proposal for: Functionality classes for random number generators. AIS 20 / AIS31, 2011.
16. Lacharme, P., Rock, A., Strubel, V., and Videau, M. The linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012.
17. Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Public keys. pp. 626–642.
18. Nguyen, P. Q., and Shparlinski, I. The insecurity of the digital signature algorithm with partially known nonces. 151–176.
19. Sahai, A., and Vadhan, S. P. A complete problem for statistical zero knowledge. *J. ACM 50*, 2 (2003), 196–249.
20. Wikipedia. /dev/random, 2004. [Online; accessed 22-July-2004].

A Proof of Theorem 1

We prove the two bounds in Theorem 1 as two separate propositions. Note that the first lower bound applies even when adversaries are restricted to just constant sequences.

Proposition 1. *For $q \geq 3$, let $\mathcal{SC}$ be a $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-secure scheduler against constant-rate adversaries running in time $t_{\mathcal{SC}}$. Then, either $t = O(q \cdot (t_{\mathcal{SC}} + \log q))$, $\varepsilon \geq 1/(q - 1/w_{\max} + 1)$, or*

$$r > \log_e q - \log_e(1/w_{\max}) - \log_e \log_e q - 1 \ ,$$

where $r = \alpha \cdot \beta$ is the competitive ratio.

Proposition 2. *Suppose that $\mathcal{SC}$ is a $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-secure scheduler running in time $t_{\mathcal{SC}}$. Then, either $t = O(q(t_{\mathcal{SC}} + \log q))$, $r^2 > w_{\max}^2 q$, $\varepsilon \geq 1/e$, or*

$$\alpha > \frac{w_{\max}}{w_{\max}+1} \cdot \frac{\log_e(1/\varepsilon) - 1}{\log_e \log_e(1/\varepsilon) + 1} \ ,$$

where $r = \alpha \cdot \beta$.

It should be clear that Theorem 1 follows immediately from the two propositions.

A.1 Proof of Proposition 1

The main step in the proof of Proposition 1 is the following lemma:

Lemma 1. *For any $q \geq 3$ let $\mathcal{E}_i$ be the constant sequence sampler that simply outputs the sequence $(1/i, \ldots, 1/i)$ for $i = 1/w_{\max}, \ldots q$. Then, for any keyless scheduler $\mathcal{SC}$ with P pools, there exists an i and an adversary $\mathcal{A}$ such that $\mathcal{E}_i$ and $\mathcal{A}$ win $\mathsf{SGAME}(P, q, w_{\max}, r)$ for any $r > \log_e q - \log_e(1/w_{\max}) - \log_e \log_e q - 1$.*

Furthermore, there exists a single adversary $\mathcal{A}'$ that, given any keyless scheduler $\mathcal{SC}$, i, and r, can output the τ that allows $\mathcal{E}_i$ to win $\mathsf{SGAME}(P, q, w_{\max}, r)$ against $\mathcal{SC}$ (or outputs `FAIL` if none exists) in time $O(q \cdot (\log q + t_{\mathcal{SC}}))$, where $t_{\mathcal{SC}}$ is the run-time of the scheduler.

Proof. We assume without loss of generality that $1/w_{\max}$ is an integer.

Fix any keyless scheduler $\mathcal{SC}$ and start state τ_0. Given the corresponding sequence $(\mathsf{in}_j, \mathsf{out}_j)_{j=1}^q$, we define the sequence of "leave times" $b_1, \ldots b_q \in \mathbb{N} \cup \{\infty\}$ as $b_j = \min\{T \geq j : \mathsf{out}_T = \mathsf{in}_j\}$ (where we adopt the convention that $\min \varnothing = \infty$). Intuitively, at time T, we imagine the scheduler selecting a pool in_T in which to "throw a ball", and a pool out_T to empty afterwards. The leave time b_j is the time at which the ball that was "thrown" at time j will "leave the game".

Let τ_T be the state of $\mathcal{SC}$ after T steps, and let $\mathcal{A}_T$ be the adversary that sets the state of $\mathcal{SC}$ to τ_T. Note that $\mathcal{SC}$ wins $\mathsf{SGAME}(P, q, w_{\max}, r)$ against $\mathcal{E}_i, \mathcal{A}_T$ if and only if there is some set of i balls $J \subseteq [T+1, T+i \cdot r]$ with $b_j = b_{j'} \leq T + i \cdot r$ for all $j, j' \in J$.

We proceed by "marking balls". We first consider $\lfloor \frac{w_{\max} \cdot q}{r} \rfloor$ non-overlapping intervals of length $r/w_{\max}$ in $\{1, \ldots, q\}$. By hypothesis, there must be at least $1/w_{\max}$ balls in each of these intervals that leave at the same time in the same interval. We mark all such balls, marking at least $\frac{q}{r} - 1/w_{\max}$ distinct balls in total. Now, consider $\lfloor \frac{w_{\max} \cdot q}{2r} \rfloor$ non-overlapping intervals of length $2r/w_{\max}$. In each such interval, there must be at least $2/w_{\max}$ balls whose leave time is the same and in the interval. We mark these balls. Previously no more than $1/w_{\max}$ balls that we'd marked had the same leave time, so we must have marked at least $1/w_{\max}$ new balls in each interval. Therefore, we've now marked at least $\frac{q}{r} + \frac{q}{2r} - 2/w_{\max}$ distinct balls, and no set of more than $2/w_{\max}$ balls have the same leave time.

Proceeding by induction, suppose that after $j < \lfloor \frac{w_{\max} \cdot q}{r} \rfloor$ steps, we have marked at least $\sum_{k=1}^{j} \frac{q}{k \cdot r} - j/w_{\max}$ distinct balls, and no set of more than $j/w_{\max}$ marked balls have the same leave time. We consider $\lfloor \frac{w_{\max} \cdot q}{(j+1) \cdot r} \rfloor$ non-overlapping intervals of length $(j+1) \cdot r/w_{\max}$ and note that in each such interval there must be $(j+1)/w_{\max}$ balls with the same leave time. So, we mark these and note that we must have marked an additional $\frac{q}{2r} - 1/w_{\max}$ new balls and that no set of more than $(j+1)/w_{\max}$ marked balls have the same leave time.

It follows that this procedure will mark at least $\sum_{k=1}^{\lfloor w_{\max}\cdot q/r\rfloor} \frac{q}{k\cdot r} - q/r$ balls. Recalling that the nth harmonic number satisfies $H_n = \sum_{k=1}^{n} 1/k > \log_e(n+1)$, it follows that we've marked at least $\frac{q}{r}\cdot(\log_e q - \log_e r - \log_e(1/w_{\max}) - 1)$ distinct balls in this way. But, there are only q balls total. It follows that $r > \log_e q - \log_e(1/w_{\max}) - \log_e \log_e q - 1$.

It remains to construct an $\mathcal{A}'$ that finds the winning τ in $O(q\cdot(t_{\mathcal{SC}} + \log q))$ time given $\mathcal{SC}$, i, and r. $\mathcal{A}'$ first computes $(\tau_j)_{j=0}^{q-1}$ in time $O(q\cdot t_{\mathcal{SC}})$. Now, as above, $\mathcal{A}'$ divides $\{1,\ldots,q\}$ into disjoint intervals of length $\lfloor\frac{q}{i\cdot r}\rfloor$. For each such interval $[T+1, T+i\cdot r]$, $\mathcal{A}'$ simply simulates $\mathsf{SGAME}(P, i\cdot r, r)$ against $\mathcal{E}_i$ starting at τ_T.[13] $\mathcal{A}$ returns τ_T if it wins the simulation. If no τ_T wins, $\mathcal{A}'$ outputs `FAIL`. This takes time $O(q\log q)$. (The $\log q$ overhead is incurred because $\mathcal{A}$ needs to write numbers that could be as large as q.)

The result follows. ◻

From this, Proposition 1 follows easily.

Proof of Proposition 1. Fix $\mathcal{SC}$.

Let $\mathcal{E}$ be the sequence sampler that selects $i \xleftarrow{\$} \{1/w_{\max},\ldots,q\}$ and then behaves as the constant sequence sampler $\mathcal{E}_i$ from Lemma 1. Let $\mathcal{A}$ be the adversary that behaves as follows: On input skey, $\mathcal{A}$ produces the keyless scheduler $\mathcal{SC}_{\mathsf{skey}}$ such that $\mathcal{SC}_{\mathsf{skey}}(\sigma) = \mathcal{SC}(\mathsf{skey}, \sigma)$. $\mathcal{A}$ then simulates $\mathcal{A}'$ from the lemma, which outputs either some state τ or `FAIL`. If $\mathcal{A}'$ outputs τ, $\mathcal{A}$ simply does the same. Otherwise, $\mathcal{A}$ outputs an arbitrary state.

By Lemma 1, $\mathcal{A}$ runs in time $O(q\cdot(\log q + t_{\mathcal{SC}}))$, and if $r \le \log_e q - \log_e(1/w_{\max}) - \log_e\log_e q - 1$, then with probability at least $1/(q - 1/w_{\max} + 1)$, this procedure produces an $\mathcal{E}_i, \tau$ pair that wins $\mathsf{SGAME}(P, q, w_{\max}, r)$ against $\mathcal{SC}_{\mathsf{skey}}$. The result follows. ◻

A.2 Proof of Proposition 2

Proof of Proposition 2. Suppose $r^2 \le w_{\max}^2 q$. For simplicity, we will assume $1/w_{\max}$ is an integer.

Our proof begins similarly to that of Lemma 1. In particular, we let τ_0 be any start state. Let $B_1,\ldots,B_q$ be random variables over the choice of skey corresponding to leave times, $B_j = \min\{T \ge j : \mathsf{out}_T = \mathsf{in}_j\}$. We again think of a ball with weight w_j thrown into pool in_j at time j and leaving the game at time B_j.

Intuitively, our approach will be to first show a pair of adversaries that win if balls take too long to leave. We'll then show a pair of adversaries that win if balls leave too quickly.

In particular, let $\mathcal{E}$ simply output a sequence of $\alpha/w_{\max}$ maximum weights followed by 0s, $(w_{\max},\ldots,w_{\max},0,\ldots 0)$. For any skey and any $1 \le T \le q$, let $\tau_T(\mathsf{skey})$ be the state that $\mathcal{SC}$ with skey reaches after T steps, starting at τ_0. Let $\mathcal{A}_k$ be the adversary that simply outputs $\tau_{kr/w_{\max}}(\mathsf{skey})$ on input skey. Note in order for $\mathcal{SC}$ to win SGAME against $\mathcal{E}, \mathcal{A}_k$, it is necessary but not sufficient for there to be some j with $kr/w_{\max} < j \le kr/w_{\max} + \alpha/w_{\max}$ and $B_j \le (k+1)r/w_{\max}$. (Intuitively, there must be some ball that enters in the first $\alpha/w_{\max}$ steps of the game against $\mathcal{A}_k$ and leaves before time $r/w_{\max}$.)

Now, let $\mathcal{A}_k^*$ be an adversary that for $0 \le k' < k$ selects $j_{k'}$ uniformly at random with $k'r/w_{\max} < j_{k'} \le k'r/w_{\max} + \alpha/w_{\max}$. If $B_{j_{k'}} > (k'+1)\cdot r/w_{\max}$, then $\mathcal{A}_k^*$ simply behaves as $\mathcal{A}_{k'}$. Otherwise, $\mathcal{A}_k^*$ behaves as $\mathcal{A}_k$. Let E_k be the event that $B_{j_{k'}} < (k'+1)\cdot r/w_{\max}$ for all $k' \le k$. Note that $\mathcal{A}_k^*$ wins if E_k happens and $B_j > (k+1)\cdot r/w_{\max}$ for all j with $kr/w_{\max} < j \le kr/w_{\max} + \alpha/w_{\max}$. (To be clear, A_k^* may win in other circumstances as well.) Therefore,

$$\varepsilon \ge \Pr[E_k]\cdot \Pr\left[\forall j \text{ with } \frac{kr}{w_{\max}} < j \le \frac{kr+\alpha}{w_{\max}},\ B_j > (k+1)\cdot\frac{r}{w_{\max}} \;\middle|\; E_k\right].$$

[13] Technically, we replace $\mathcal{E}_i$ with $\mathcal{E}_i'$, which outputs a sequence of length $i\cdot r$.

Rearranging, we have

$$\begin{aligned}
\Pr[E_k] - \varepsilon &\le \Pr[E_k] \cdot \Pr\left[\exists j \text{ with } \frac{kr}{w_{\max}} < j \le \frac{kr+\alpha}{w_{\max}},\ B_j \le (k+1)\cdot \frac{r}{w_{\max}} \;\middle|\; E_k\right] \\
&\le \Pr[E_k] \cdot \sum_{j=k\cdot r/w_{\max}}^{(k\cdot r+\alpha)/w_{\max}} \Pr\left[B_j \le (k+1)\cdot \frac{r}{w_{\max}} \;\middle|\; E_k\right] \\
&= \frac{\alpha}{w_{\max}} \cdot \Pr[E_k] \cdot \Pr\left[B_{j_k} \le (k+1)\cdot \frac{r}{w_{\max}} \;\middle|\; E_k\right] \\
&= \frac{\alpha}{w_{\max}} \cdot \Pr[E_{k+1}] \ ,
\end{aligned}$$

where B_{j_k} is chosen uniformly at random with $kr/w_{\max} < j_k \le kr/w_{\max} + \alpha/w_{\max}$. So, we have the recurrence relation $\Pr[E_k] \ge (w_{\max}/\alpha) \cdot (\Pr[E_{k-1}] - \varepsilon)$, with $\Pr[E_0] = 1$. It follows that

$$\Pr[E_k] \ge \Big(\frac{w_{\max}}{\alpha}\Big)^k - \varepsilon \cdot \sum_{i=1}^{k} \Big(\frac{w_{\max}}{\alpha}\Big)^i > \Big(\frac{w_{\max}}{\alpha}\Big)^k - \varepsilon \cdot \frac{w_{\max}}{\alpha - w_{\max}} \ .$$

Now, let $\mathcal{E}^*$ be the sequence sampler that randomly selects j_k with $kr/w_{\max} < j_k \le kr/w_{\max} + \alpha/w_{\max}$ for all $k < (w_{\max}+1)\cdot \alpha/w_{\max}$. $\mathcal{E}^*$ then outputs the sequence (w_i) where $w_i = w_{\max}/(w_{\max}+1)$ if $i = j_k$ for some k and $w_i = 0$ otherwise. Suppose the event E_{k^*} occurs where $k^* = (w_{\max}+1)\cdot(\alpha-1)/w_{\max} + 1$. Then, for all $k \le k^*$, the j_k-th ball leaves before the j_{k+1}-st ball enters. In particular, $\mathcal{E}^*, \mathcal{A}_0$ win SGAME. Therefore,

$$\varepsilon \ge \Pr[E_{k^*}] > \Big(\frac{w_{\max}}{\alpha}\Big)^{k^*} - \varepsilon \cdot \frac{w_{\max}}{\alpha - w_{\max}} \ .$$

It follows that

$$\alpha > \frac{w_{\max}}{w_{\max}+1} \cdot \frac{\log_e(1/\varepsilon) - 1}{\log_e \log_e(1/\varepsilon) + 1}$$

provided that $\varepsilon < 1/e$.

It is easy to see that $\mathcal{A}_k^*$ and $\mathcal{E}^*$ run in time $O(q(t_{\mathcal{SC}} + \log q))$, and the result follows. □

B Construction of Constant-Rate Scheduler and Proof of Theorem 5

We first notice that Fortuna's scheduler can be easily modified to use a different base. In particular, for any integer $b \ge 2$, we define a keyless scheduler, $\mathcal{SC}_b$. Roughly, $\mathcal{SC}_b$ has $P_b \approx \log_b q$ pools, numbered $0, \ldots, P_b - 1$. The state $\tau \in \{0, \ldots, q-1\}$ will just be a counter. The pools are filled in turn, and pool i is emptied whenever the counter τ divides $b^i \cdot P_b$ but not $b^{i+1} \cdot P_b$.

Our actual construction will be slightly more involved than the above, but it is simply an optimized version of this basic idea. In particular, we make four changes:

1. We account for $w_{\max}$ by emptying pools when τ divides $b^i \cdot P_b/w_{\max}$, instead of just $b^i \cdot P_b$.
2. We use slightly fewer than $\log_b q$ pools, setting $P_b = \log_b q - \log_b \log_b q - \log_b(1/w_{\max})$.
3. We do not empty the 0th pool twice in a row. (While this never comes up when $b = 2$, it is an issue for $b \ge 3$.)
4. If pool j will next be emptied sooner than pool i and $j > i$, we fill pool j instead of pool i. (This captures the idea of emptying multiple pools at once from Section 6.)

For simplicity, we assume that $\log_b \log_b q$ and $\log_b(1/w_{\max})$ are both integers, and we let $P_b = \log_b q - \log_b \log_b q - \log_b(1/w_{\max})$. Then, we define $\mathcal{SC}_b$ as in Figure 10.

Theorem 5 shows that this scheme achieves a very good competitive ratio of $r_b \approx bP_b$. In Appendix A, we show a lower bound in the constant-rate case of $r > \log_e q - \log_e \log_e q - \log_e(1/w_{\max}) - 1$ (or $r > P_e - 1$ in slightly abused notation), so this result is very close to optimal.

proc. $\mathcal{SC}_b(\tau)$:
`IF` $\tau \neq 0 \mod P_b/w_{\max}$, `THEN` $\mathsf{out} \leftarrow \bot$
`ELSE`
    $j \leftarrow \max\{j : \tau = 0 \mod b^j \cdot P_b/w_{\max}\}$
    `IF` $j = 0$ `AND` $\tau - P_b/w_{\max} \neq 0 \mod b \cdot P_b/w_{\max}$
        $\mathsf{out} \leftarrow \bot$ // *Don't empty pool* 0 *twice in a row*
    `ELSE` $\mathsf{out} \leftarrow j$
$i \leftarrow \tau - 1 \mod P_b$
// *Find the next time* τ^* *that a pool with index at least* i *will be emptied*
$\tau^* \leftarrow \min\{\tau^* \geq \tau : \tau^* = 0 \mod b^i \cdot P_b/w_{\max}\}$
// *Fill the pool emptied at time* τ^*
$\mathsf{in} \leftarrow \max\{\mathsf{in} : \tau^* = 0 \mod b^{\mathsf{in}} \cdot P_b/w_{\max}\}$
$\tau' \leftarrow \tau + 1 \mod q$
`OUTPUT` $(\tau', \mathsf{in}, \mathsf{out})$

Fig. 10: Our keyless scheduler construction

Proof of Theorem 5. Note that $\mathcal{E}$ must output a constant sequence, $(w, \ldots, w)$ with $r_b/q \leq w \leq w_{\max}$. (If $w < r_b/q$, then we win by default.) We assume without loss of generality that $1/w$ is an integer.

We first handle the case when $w > w_{\max}/b$. Note that no pool is emptied more than once every $\frac{b}{w_{\max}} \cdot P_b$ steps and at least one pool is emptied every $\frac{b-1}{w_{\max}} \cdot P_b$ steps. So, if $w > w_{\max}/b$, $\mathcal{SC}_b$ wins as soon as the first pool is emptied after $\frac{1}{w} \cdot P_b$ steps, in time at most $(\frac{1}{w} + \frac{b-1}{w_{\max}}) \cdot P_b$. It therefore achieves a competitive ratio of less than $(1 + (b-1) \cdot \frac{w}{w_{\max}}) \cdot P_b \leq bP_b$.

Now, assume $w \leq w_{\max}/b$.

Let $i \geq 1$ such that $\frac{b^{i+1}-1}{b-1} \geq w_{\max}/w > \frac{b^i-1}{b-1}$. Consider the first time a pool whose index is at least i is emptied. If it is full on this first emptying, then $\mathcal{SC}_b$ wins, in time at most $b^i \cdot P_b/w_{\max}$. Otherwise, let T^* be the first time such a pool is emptied. Then, $\mathcal{SC}_b$ wins the next time a pool whose index is greater than i is emptied, at time $T^* + b^i \cdot P_b/w_{\max}$. In both cases, $\mathcal{SC}_b$ achieves a competitive ratio of at worst $r_b = w \cdot (T^* + b^i \cdot P_b/w_{\max})$.

We wish to bound T^*. Let j such that $b^{j+1} > w_{\max} \cdot T^*/P_b \geq b^j$. Then, at time T^* the pool that is emptied has weight at least

$$\begin{aligned}
w \cdot \lfloor T^*/P_b \rfloor + \frac{w}{w_{\max}} \cdot \sum_{k=0}^{j} b^k &> w \cdot \left(\frac{T^*}{P_b} + \frac{1}{w_{\max}} \cdot \frac{b^{j+1}-1}{b-1} - 1\right) \\
&> w \cdot \left(\frac{T^*}{P_b} + \frac{1}{w_{\max}} \cdot \frac{w_{\max} \cdot \frac{T^*}{P_b} - 1}{b-1} - 1\right) \\
&= \frac{w \cdot T^*}{P_b} \cdot \frac{b}{b-1} - w \cdot \frac{1 + (b-1) \cdot w_{\max}}{(b-1) \cdot w_{\max}} .
\end{aligned}$$

Note that the above weight is less than one by hypothesis. Applying this and rearranging,

$$w \cdot T^* < \frac{w + (1+w)(b-1) \cdot w_{\max}}{b \cdot w_{\max}} \cdot P_b .$$

Plugging in and recalling that $w \leq w_{\max}/b$ and $w_{\max}/w > \frac{b^i-1}{b-1}$,

$$\begin{aligned}
r_b &< \frac{w + (1+w)(b-1) \cdot w_{\max}}{b \cdot w_{\max}} \cdot P_b + \frac{w}{w_{\max}} \cdot \left((b-1) \cdot \frac{w_{\max}}{w} + 1\right) \cdot P_b \\
&\leq \left(\frac{(1 + w_{\max}/b)(b-1)}{b} + \frac{1}{b^2}\right) \cdot P_b + \left(b - 1 + \frac{1}{b}\right) \cdot P_b \\
&= \left(b + \frac{w_{\max}}{b} + \frac{1 - w_{\max}}{b^2}\right) \cdot P_b
\end{aligned}$$

The result follows.

□

C Recovering and Preserving Secutity

C.1 Recovering Security

We consider the following security game with an attacker $\mathcal{A}$, a sampler $\mathcal{D}$, and bounds $q_{\mathcal{D}}, \gamma^*$.

- $\mathcal{D}$ sends $J \subset \{1, \ldots, q_{\mathcal{D}}\}$ to the challenger.
- The challenge chooses a seed $\mathsf{seed} \overset{\$}{\leftarrow} \mathsf{setup}$, and a bit $b \overset{\$}{\leftarrow} \{0,1\}$ uniformly at random. It sets $\sigma_0 := 0$. For $k = 1, \ldots, q_{\mathcal{D}}$, the challenger computes

$$(\sigma_k, I_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1}).$$

- The attacker $\mathcal{A}$ gets seed, J, and $\gamma_1, \ldots, \gamma_{q_{\mathcal{D}}}, z_1, \ldots z_{q_{\mathcal{D}}}$. It gets access to an oracle $\mathsf{get\text{-}refresh}()$ which initially sets $k := 0$ on each invocation increments $k := k+1$ and outputs I_k. At some point the attacker $\mathcal{A}$ outputs a value $S_0 \in \{0,1\}^n$, an integer d, and I_j^* for $j \in J$ such that $k + d \leq q_{\mathcal{D}}$ and

$$\sum_{\substack{k<j\leq k+d \\ j \notin J}} \gamma_j \geq \gamma^* \ .$$

- For $j = k+1, \ldots, k+d$, the challenger computes

$$S_j \leftarrow \begin{cases} \mathsf{refresh}(S_{j-1}, I_j) : j \notin J \\ \mathsf{refresh}(S_{j-1}, I_j^*) : j \in J \end{cases} .$$

  If $b = 0$ it sets $(S^*, R) \leftarrow \mathsf{next}(S_d)$ and if $b = 1$ is sets $(S^*, R) \leftarrow \{0,1\}^{n+\ell}$ uniformly at random. The challenger gives $I_{k+d+1}, \ldots, I_{q_{\mathcal{D}}}$, and (S^*, R) to $\mathcal{A}$.
- The attacker $\mathcal{A}$ outputs a bit b^*.

Definition 6 (Recovering Security). *We say that PRNG with input has $(t, q_{\mathcal{D}}, \gamma^*, \varepsilon)$-*recovering security *if for any attacker $\mathcal{A}$ and legitimate sampler $\mathcal{D}$, both running in time t, the advantage of the above game with parameters $q_{\mathcal{D}}, \gamma^*$ is at most ε.*

C.2 Preserving Security

We define preserving security exactly as in [5]. Intuitively, it says that if the state S_0 starts uniformly random and uncompromised and is then refreshed with arbitrary (adversarial) samples $I_1, \ldots, I_d$ resulting in some final state S_d, then the output $(S^*, R) \leftarrow \mathsf{next}(S_d)$ looks indistinguishable from uniform.

- The challenger chooses an initial state $S_0 \leftarrow \{0,1\}^n$, a seed $\mathsf{seed} \leftarrow \mathsf{setup}$, and a bit $b \leftarrow \{0,1\}$ uniformly at random.
- The attacker $\mathcal{A}$ gets seed and specifies an arbitrarily long sequence of values $I_1, \ldots, I_d$ with $I_j \in \{0,1\}^n$ for all $j \in [d]$.
- The challenger sequentially computes

$$S_j = \mathsf{refresh}(S_{j-1}, I_j, \mathsf{seed})$$

  for $j = 1, \ldots, d$. If $b = 0$ the attacker is given $(S^*, R) = \mathsf{next}(S_d)$ and if $b = 1$ the attacker is given $(S^*, R) \leftarrow \{0,1\}^{n+\ell}$.
- The attacker outputs a bit b^*.

Definition 7 (Preserving Security). *A PRNG with input has (t, ε)-*preserving security *if for any attacker $\mathcal{A}$ running in time t, the advantage of $\mathcal{A}$ in the above game is at most ε.*

C.3 Modified Composition Theorem

With these modified definitions, [5]'s proof of their composition theorem immediately extends to handle semi-adaptive set-refresh queries.

Theorem 8. *Assume that a PRNG with input has both (t, ε_p)-preserving security and $(t, q_{\mathcal{D}}, \gamma^*, \varepsilon_r)$-recovering security as defined above. Then, it is $((t', q_{\mathcal{D}}, q_R, q_S), \gamma^*, q_R(\varepsilon_r + \varepsilon_p))$-robust in the semi-adaptive* set-refresh *model where $t' \approx t$.*