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. We study random number generators (RNGs) with input, RNGs that regularly update their internal state according to some auxiliary input with additional randomness harvested from the environment. We formalize the problem of designing an efficient recovery mechanism from complete state compromise in the presence of an active attacker. 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*. 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 is that any entropy used prematurely will be lost, and any entropy which is kept unused will delay the recovery.

After formally modeling RNGs with input, we show a nearly optimal construction that is secure in our very strong 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.

Keywords: Random number generators, RNGs with input

⋆ 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.

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 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` [21] for Linux, Yarrow [14] for MacOs/iOS/FreeBSD and Fortuna [10] for Windows [9].

Unfortunately, despite the fact that they are widely used and often referred to in various standards [2, 8, 13, 16], 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 [15], 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 [10,20] (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 [14] and used by MacOS/iOS/FreeBSD), and its refinement Fortuna (subsequently designed by Ferguson and Schneier [10] and used by Windows [9]). 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 5.3, 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 the full version for details [6]), which is comparable to [10]'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 the full version [6], 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 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 the full version 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 [6].

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 [17]. On the other hand, many works showed devastating attacks on various cryptographic schemes when using weak randomness; some notable examples include [4, 7, 11, 12, 15, 18, 19].

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 initialize procedure, procedures to respond to adversary oracle queries, and a finalize procedure. A game GAME is executed with an adversary $\mathcal{A}$ as follows: First, 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 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 $\mathsf{Adv}^{\mathsf{GAME}}_\mathcal{A} = 2 \times \Pr[\mathsf{GAME}^\mathcal{A} \Rightarrow 1] - 1$. 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* $S' = \mathsf{refresh}(\mathsf{seed}, S, I) \in \{0,1\}^n$*.*
- next*: a deterministic algorithm that, given* seed *and a state* $S \in \{0,1\}^n$*, outputs a pair* $(S', R) = \mathsf{next}(\mathsf{seed}, S)$ *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

$$\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_{\mathcal{D}}}, z_1, \ldots, z_{q_{\mathcal{D}}}, \gamma_0, \ldots, \gamma_{q_{\mathcal{D}}}) \geq \gamma_j \qquad (1)$$

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 $\sigma_0 = 0$.[4]

Dodis et al. provide a detailed discussion of the distribution sampler in [5], which we also include in the full version of this paper for completeness [6]. In particular, note that the distribution sampler $\mathcal{D}$ is required to output a lower bound γ on the min-entropy of I. These entropy estimates will be used in the security game. In particular, we of course cannot guarantee security unless the distribution sampler has provided the challenger with some minimum amount of entropy. Many implemented RNGs try to get around this problem by attempting to estimate the entropy of a given distribution directly in some ad-hoc manner. However, entropy estimation is impossible in general and computationally hard even in very special cases [20]. Note that these entropy estimates will be used only in the security game, and are *not* given to the refresh and next procedures. By separating entropy estimation from security, [5] provides a meaningful definition of security without requiring the RNG to know anything about the entropy of the sampled distributions.

3.2 Security Notions

We define the game $\mathsf{ROB}(\gamma^*)$ in our game framework. We show the initialize and 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 (get-state and set-state), to see output from it (get-next), and to update

[4] 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.

it with a sample I_j from $\mathcal{D}$ ($\mathcal{D}$-refresh). In particular, notice that the $\mathcal{D}$-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 initialize and 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. get-state
$c \leftarrow 0$; corrupt $\leftarrow$ true
OUTPUT S

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

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

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* $\mathcal{D}$-refresh*,* q_R *calls to* next-ror *or* get-next *and* q_S *calls to* get-state *or* set-state*, and any legitimate distribution sampler* $\mathcal{D}$ *inside the* $\mathcal{D}$-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 get-next when the RNG is still corrupted, this is a "premature" get-next and the entropy counter c is reset to 0. Intu-

itively, [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 initialize and 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" get-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 $\mathcal{D}$-refresh since the last set-state or get-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 get-state or set-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.

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 *or* get-next *and* q_S *calls to* get-state *or* 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:

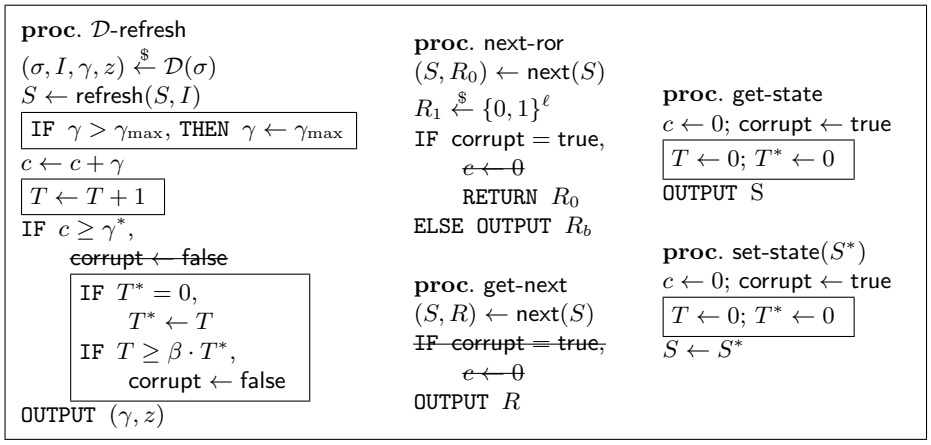

Fig. 3: Procedures in $\mathsf{NROB}(\gamma^*, \gamma_{\max}, \beta)$ for $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ with differences from $\mathsf{ROB}(\gamma^*)$ highlighted

- $\mathsf{NROB}_{\mathsf{reset}}(\gamma^*, \gamma_{\max}, \beta)$ 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.[5]
- $\mathsf{NROB}_{\mathsf{1set}}(\gamma^*, \gamma_{\max}, \beta)$ 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 about the (low-entropy) state. Surprisingly, Ferguson and Schneier presented an elegant away to get around this issue in their Fortuna construction [10]. 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

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

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,* $\mathsf{in}, \mathsf{out} \in \mathbb{N} \cup \{\perp\}$.

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 $\mathsf{in}, \mathsf{out} \in [0, P-1] \cup \{\perp\}$.

Given a scheduler $\mathcal{SC}$ with skey and state τ, we write

$$\mathcal{SC}^q(\mathsf{skey}, \tau) = (\mathsf{in}_j(\mathcal{SC}, \mathsf{skey}, \tau), \mathsf{out}_j(\mathcal{SC}, \mathsf{skey}, \tau))_{j=1}^q \tag{2}$$

to represent the sequence obtained by computing $(\mathsf{in}, \mathsf{out}, \tau) \leftarrow \mathcal{SC}(\mathsf{skey}, \tau)$ repeatedly, a total of 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 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 $w_{\max} = \gamma_{\max}/\gamma^*$.

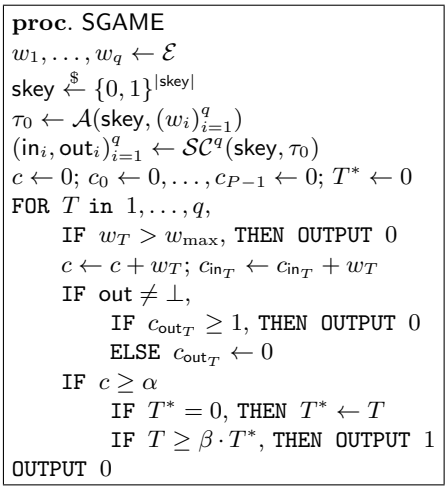

Fig. 4: $\mathsf{SGAME}(P, q, w_{\max}, \alpha, \beta)$, the security game for a scheduler $\mathcal{SC}$

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 $\sum_{i=1}^{T^*} w_i \geq \alpha$ 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 $\mathcal{SC}$ is* $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-secure *if it has P pools and 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 $\mathcal{SC}$.*[6]

We note that schedulers are non-trivial objects. Indeed, in the full version of the paper [6], we prove the following lower bounds, which in particular 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 $q \geq 3$, $\varepsilon < 1/q$,*

[6] 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.

$t = \Omega(q \cdot (t_{\mathcal{SC}} + \log q))$, *and* $r \leq w_{\max}\sqrt{q}$, *we have that*

$$r > \log_e q - \log_e(1/w_{\max}) - \log_e \log_e q - 1 \ , \tag{3}$$

$$\alpha > \frac{w_{\max}}{w_{\max}+1} \cdot \frac{\log_e(1/\varepsilon) - 1}{\log_e \log_e(1/\varepsilon) + 1} \ . \tag{4}$$

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 use only ρ.

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.

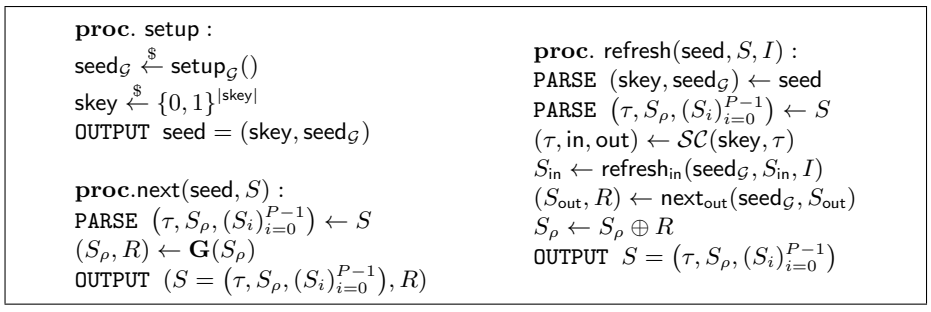

Fig. 5: The generalized Fortuna construction

We prove the following composition theorem in the full version of this paper [6].

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* $\mathsf{NROB}_{\mathsf{0set}}$, $\varepsilon' = q_{\mathcal{D}} \cdot \varepsilon_{\mathcal{SC}} + P \cdot \varepsilon + q'_R \varepsilon_{\mathsf{prg}}$.

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 [10]. 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 $q_{\mathcal{D}} = q_R = q_S = 2^{32}$.

In this setting, guidelines of [5, Section 6.1] show that their 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

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 [10], 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

proc. $\mathcal{SC}(\mathsf{skey}, \tau)$:
IF $\tau \neq 0 \mod P/w_{\max}$, THEN $\mathsf{out} \leftarrow \perp$
ELSE $\mathsf{out} \leftarrow \max\{\mathsf{out} : \tau = 0 \mod 2^{\mathsf{out}} \cdot P/w_{\max}\}$
$\mathsf{in} \leftarrow \mathbf{F}(\mathsf{skey}, \tau)$
$\tau' \leftarrow \tau + 1 \mod q$
OUTPUT $(\tau', \mathsf{in}, \mathsf{out})$

Fig. 6: Our scheduler construction

is a counter τ. The pools are filled in a "round-robin" fashion. 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 $w_i = w$.[7] 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.

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 6. 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.)

The following theorem is proven in the full version of this paper [6].

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* $t' \approx t$*,* $\varepsilon_{\mathcal{SC}} = q \cdot (\varepsilon_{\mathbf{F}} + \varepsilon)$*,* $\alpha = 2 \cdot (w_{\max} \cdot \log_e(1/\varepsilon) + 1) \cdot (\log_2 q - \log_2 \log_2 q - \log_2(1/w_{\max}))$*, and* $\beta = 4$*.*

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. So, one can safely instantiate our scheduler (and the corresponding RNG) without a known bound on* $w_{\max}$*, and still benefit if* $w_{\max}$ *happens to be low in practice.*

Instantiation and Concrete Numbers. 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 $\varepsilon_{\mathcal{SC}} \approx q\varepsilon$.[8] 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:

[7] We analyze their construction against constant sequences more carefully in Section 6.
[8] See [5] for justification for such an assumption.

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

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

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

5.3 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 5.3 are not ideal. But, our security model is also very strong. So, we follow Ferguson and Schneier [10] and consider the weaker model in which the distribution sampler $\mathcal{D}$ is restricted to a constant entropy rate. Analysis in this model suggests that our construction may perform much in practice. Indeed, if we think of the distribution sampler $\mathcal{D}$ as essentially representing nature, this model may not be too unreasonable.

Constant-Rate Model. We simply modify our definitions in the natural way. 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). We say that a scheduler is $(t, q, w_{\max}, r, \varepsilon)$-*secure against constant sequences* if, for some[9] α, β such that $\alpha \cdot \beta = r$ it is $(t, q, w_{\max}, \alpha, \beta, \varepsilon)$-secure when the 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. We similarly define premature-next robustness for RNGs with input.

In the full version [6], we the note that our composition theorem, Theorem 2, applies equally well in the constant-rate case. This allows us to construct an RNG with input that is premature-next robust against constant adversaries with much better parameters.

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 [10]. (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

[9] 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 $\alpha \cdot \beta = \alpha' \cdot \beta'$.

$3 \log_2 q$.[10] 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.[11]

Interestingly, the pseudocode in [10] 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 $j \leq i$. 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. While our model cannot syntactically capture the notion of multiple pools being emptied at once, we notice that it 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 the full version of this paper, we make this observation concrete and further optimize the scheduler of Fortuna to obtain the following result [6].

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

$$r_b = \left(b + \frac{w_{\max}}{b} + \frac{1 - w_{\max}}{b^2}\right) \cdot (\log_b q - \log_b \log_b q - \log_b(1/w_{\max})) \; . \tag{5}$$

In particular, with $w_{\max} = 1$ and $q \to \infty$, $b = 3$ is optimal with

$$r_3 \approx 2.1 \log_2 q \approx \frac{r_{\mathcal{F}}}{1.66} \approx \frac{r_2}{1.19} \approx \frac{r_4}{1.01} \; . \tag{6}$$

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.

References

1. Barak, B., and Halevi, S. A model and architecture for pseudo-random generation with applications to /dev/random. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, 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.

[10] 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 $3 \log_2 q$.

[11] This follows from the analysis of our own scheduler in the full version [6].

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. Dodis, Y., Shamir, A., Stephens-Davidowitz, N., and Wichs, D. How to eat your entropy and have it too – optimal recovery strategies for compromised rngs. Cryptology ePrint Archive, Report 2014/167, 2014. `http://eprint.iacr.org/`.
7. 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.
8. Eastlake, D., Schiller, J., and Crocker, S. *RFC 4086 - Randomness Requirements for Security*, June 2005.
9. Ferguson, N. Private communication, 2013.
10. Ferguson, N., and Schneier, B. *Practical Cryptography*, 1 ed. John Wiley & Sons, Inc., New York, NY, USA, 2003.
11. Gutterman, Z., Pinkas, B., and Reinman, T. Analysis of the linux random number generator. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), SP '06, IEEE Computer Society, pp. 371–385.
12. 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).
13. Information technology - Security techniques - Random bit generation. ISO/IEC18031:2011, 2011.
14. 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.
15. 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.
16. Killmann, W. and Schindler, W. A proposal for: Functionality classes for random number generators. AIS 20 / AIS31, 2011.
17. Lacharme, P., Röck, A., Strubel, V., and Videau, M. The linux pseudorandom number generator revisited. *IACR Cryptology ePrint Archive 2012* (2012), 251.
18. Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Public keys. pp. 626–642.
19. Nguyen, and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology 15*, 3 (2002), 151–176.
20. Sahai, A., and Vadhan, S. P. A complete problem for statistical zero knowledge. *J. ACM 50*, 2 (2003), 196–249.
21. Wikipedia. /dev/random. `http://en.wikipedia.org/wiki//dev/random`, 2004. [Online; accessed 09-February-2014].