Security Analysis of Pseudo-Random Number Generators with Input: `/dev/random` is not Robust

Yevgeniy Dodis[1], David Pointcheval[2], Sylvain Ruhault[3], Damien Vergnaud[2], and Daniel Wichs[4]

[1] Dept. of Computer Science, New York University.
[2] DI/ENS, ENS-CNRS-INRIA.
[3] DI/ENS, ENS-CNRS-INRIA and Oppida, France.
[4] Dept. of Computer Science, Northeastern University.

Abstract. A pseudo-random number generator (PRNG) is a deterministic algorithm that produces numbers whose distribution is indistinguishable from uniform. A formal security model for PRNGs with input was proposed in 2005 by Barak and Halevi (BH). This model involves an internal state that is refreshed with a (potentially biased) external random source, and a cryptographic function that outputs random numbers from the continually internal state. In this work we extend the BH model to also include a new security property capturing how it should accumulate the entropy of the input data into the internal state after state compromise. This property states that a good PRNG should be able to eventually recover from compromise even if the entropy is injected into the system at a very slow pace, and expresses the real-life expected behavior of existing PRNG designs. Unfortunately, we show that neither the model nor the specific PRNG construction proposed by Barak and Halevi meet this new property, despite meeting a weaker robustness notion introduced by BH. From a practical side, we also give a precise assessment of the security of the two Linux PRNGs, `/dev/random` and `/dev/urandom`. In particular, we show several attacks proving that these PRNGs are not robust according to our definition, and do not accumulate entropy properly. These attacks are due to the vulnerabilities of the entropy estimator and the internal mixing function of the Linux PRNGs. These attacks against the Linux PRNG show that it does not satisfy the "robustness" notion of security, but it remains unclear if these attacks lead to actual exploitable vulnerabilities in practice. Finally, we propose a simple and very efficient PRNG construction that is provably robust in our new and stronger adversarial model. We present benchmarks between this construction and the Linux PRNG that show that this contruction is on average more efficient when recovering from a compromised internal state and when generating cryptographic keys. We therefore recommend to use this construction whenever a PRNG with input is used for cryptography.

Keywords: Randomness; Entropy; Security models; `/dev/random`

1 Introduction

Generating random numbers is an essential task in cryptography. Random numbers are necessary not only for generating cryptographic keys, but are also needed in steps of cryptographic algorithms or protocols (*e.g.* initialization vectors for symmetric encryption, password generation, nonce generation, ...). Cryptography practitioners usually assume that parties have access to perfect randomness. However, quite often this assumption is not realizable in practice and random bits in protocols are generated by a *Pseudo-Random Number Generator* (PRNG). When this is done, the security of the scheme depends of course in a crucial way on the quality of the (pseudo-)randomness generated. If a user has access to a truly random bit-string, he can use a *deterministic* (or *cryptographic*) PRNG to expand this short *seed* into a longer sequence which distribution is indistinguishable from the uniform distribution to a computationally-bounded adversary (which does not know the seed). However, in many situations, it is unrealistic to assume that users have access to secret and perfect randomness. In a PRNG with input, one only assumes that users can store a secret internal state and have access to a (potentially biased) random source.

In spite of being widely deployed in practice, PRNGs with input were only formalized by Barak and Halevi in 2005 [BH05]. They proposed a security notion, called *robustness*, to capture the fact that the bits generated should look random to an observer with (partial) knowledge of the internal state and (partial) control of the entropy source. Combining theoretical and practical analysis of PRNGs with input, this paper presents an extension of the Barak-Halevi security model and analyses the Linux PRNGs `/dev/random` and `/dev/urandom`.

Randomness weaknesses in cryptography. The lack of insurance about the generated random numbers can cause serious damages in cryptographic protocols, and vulnerabilities can be exploited by attackers. One striking example is the recent failure in the Debian Linux distribution [CVE08], where a commented code in the OpenSSL PRNG with input led to insufficient entropy gathering and then to concrete attacks on TLS and SSH protocols. More recently, Lenstra, Hughes, Augier, Bos, Kleinjung and Wachter [LHA+12] showed that a non-negligible percentage of RSA keys share prime factors. Heninger, Durumeric, Wustrow and Halderman [HDWH12] presented an analysis of the behavior of Linux PRNG that explains the generation of low entropy keys when these keys are generated at boot time. Moreover, cryptographic algorithms are highly vulnerable to weaknesses in the underlying random number generation process. For instance, several works demonstrated that if nonces for DSS signature algorithm are generated with a weak pseudo-random number generator then the secret key can be quickly recovered after seeing a few signatures (see [NS02] and references therein). This illustrates the need for precise evaluation of PRNGs based on clear security requirements.

Security Models. Descriptions of PRNGs with input are given in various standards [Kil11,ISO11,ESC05]. They identified the following core components: the *entropy source* which is the source of randomness used by the generator to update an *internal state* which consists of all the parameters, variables, and other stored values that the PRNG uses for its operations.

Several desirable security properties for PRNGs with input have been identified in various standards ([ISO11,Kil11,ESC05,BK12]). These standards consider adversaries with various means: those who have access to the output of the generator; those who can control (partially or totally) the source of the generator and those who can control (partially or totally) the internal state of the generator (and combination of them). Several security notions have been defined:

– *Resilience*: an adversary must not be able to predict future PRNG outputs even if he can influence the entropy source used to initialize or refresh the internal state of the PRNG;
– *Forward security (*resp. backward security*)*: an adversary must not be able to predict past (*resp.* future) outputs even if he can compromise the internal state of the PRNG.

Desai, Hevia and Yin [DHY02] modelled a PRNG as an iterative algorithm and formalized the above security properties in this context. Barak and Halevi [BH05] model a PRNG with input as a pair of algorithms (refresh, next) and define a new security property called *robustness* that implies resilience, forward and backward security. This property actually assesses the behavior of a PRNG after compromise of its internal state and responds to the guidelines for developing PRNG given by Kelsey, Schneier, Wagner and Hall [KSWH98].

Linux PRNG. In Unix-like operating systems, a PRNG with input was implemented for the first time for Linux 1.3.30 in 1994. The entropy source comes from device drivers and other sources such as latencies between user-triggered events (keystroke, disk I/O, mouse clicks, ...). It is gathered into an internal state called the *entropy pool*. The internal state keeps an estimate of the number of bits of entropy in the internal state and (pseudo-)random bits are created from the special files `/dev/random` and `/dev/urandom`. Barak and Halevi [BH05] discussed briefly the PRNG `/dev/random` but its conformity with their robustness security definition is not formally analyzed.

The first security analysis of these PRNGs was given in 2006 by Gutterman, Pinkas and Reinman [GPR06]. It was completed in 2012 by Lacharme, Röck, Strubel and Videau [LRSV12]. Gutterman *et al.* [GPR06] presented an attack based on kernel version 2.6.10 for which a fix has been published in the following versions. Lacharme *et al.* [LRSV12] gives a detailed description of the operations of the PRNG and provides a result on the entropy preservation property of the mixing function used to refresh the internal state.

Our Contributions. From a theoretical side, we propose a new formal security model for PRNGs with input, which encompasses all previous security notions [BH05]. This new property captures how a PRNG with input should accumulate the entropy of the input data into the internal state. This property was not initially formalized in [BH05] but it actually expresses the real expected behavior of a PRNG after a state compromise, where it is expected that the PRNG quickly recovers enough entropy.

On a practical side, we give a precise assessment of the security of the two Linux PRNGs, `/dev/random` and `/dev/urandom`. In particular, we prove that these PRNGs are not robust and do not accumulate entropy properly. These properties are due to the behavior of the entropy estimator and the internal mixing function of the Linux PRNGs. We also analyze the PRNG with input proposed by Barak and Halevi. This scheme was proven robust in [BH05] but we prove that it does not generically satisfy our expected property of entropy accumulation. On the positive side, we propose a PRNG construction that is robust in the standard model and in our new stronger adversarial model.

Finally, we propose benchmarks between our PRNG construction and the Linux PRNG and we show that our PRNG construction is far more efficient than the Linux PRNG to recover from a compromised internal state and to generate cryptographic keys.

2 Preliminaries

Probabilities. When X is a distribution, or a random variable following this distribution, we denote $x \xleftarrow{\$} X$ when x is sample according to X. We denote by $M(X)$ the distribution probability of the output of the Turing machine M, while running on the input x drawn according to X, and with its random coins (if any). The notation $X \leftarrow Y$ says that X is assigned with the value of the variable Y, and that X is a random variable with distribution equal to that of Y. For a variable X and a set S (e.g., $\{0,1\}^m$ for some integer m), the notation $X \xleftarrow{\$} S$ denotes both assigning X a value uniformly chosen from S and letting X be a uniform random variable over S. The uniform distribution over n bits is denoted $\mathcal{U}_n$.

Indistinguishability. Two distributions X and Y are said (t,ε)-*computationally indistinguishable*, (that we denote $\mathbf{CD}_t(X,Y) \le \varepsilon$), if for any distinguisher $\mathcal{A}$ running within time t, $\Pr[\mathcal{A}(X)=1] - \Pr[\mathcal{A}(Y)=1] \le \varepsilon$. When $t=\infty$, meaning $\mathcal{A}$ is unbounded, we say that X and Y are ε-*close*, and their *statistical distance* is at most ε: $\mathbf{SD}(X,Y) \le \varepsilon$. $\mathbf{SD}(X,Y|Z) \le \varepsilon$ (resp. $\mathbf{CD}_t(X,Y|Z) \le \varepsilon$) is a shorthand for $\mathbf{SD}((X,Z),(Y,Z)) \le \varepsilon$ (resp. $\mathbf{CD}_t((X,Z),(Y,Z)) \le \varepsilon$.

Entropy. For a discrete distribution X over a set S, we denote its *min-entropy* by

$$\mathbf{H}_\infty(X) = \min_{x \xleftarrow{\$} X} \{-\log \Pr[X=x]\} \tag{1}$$

A distribution X is called a k-*source* if $\mathbf{H}_\infty(X) \ge k$. We also define worst-case min-entropy of X conditioned on another random variable Z by:

$$\mathbf{H}_\infty(X|Z) = -\log\left(\left[\max_{x,z} \Pr[X=x|Z=z]\right]\right) \tag{2}$$

It is worth noting that conditional min-entropy is defined more conservatively than usual, so that it satisfies the following relations (the first of which, called the *chain rule*, is not true for the "average-case" variant of conditional min-entropy):

$$\mathbf{H}_\infty(X,Z) - \mathbf{H}_\infty(Z) \ge \mathbf{H}_\infty(X|Z) \ge \mathbf{H}_\infty(X,Z) - |Z| \ge \mathbf{H}_\infty(X) - |Z| \tag{3}$$

where $|Z|$ is the bit-length of Z.

Extractors. Let $\mathcal{H} = \{h_X : \{0,1\}^n \to \{0,1\}^m\}_{X \in \{0,1\}^d}$ be a hash function family. We say that $\mathcal{H}$ is a (k,ε)-*extractor* if for any random variable I over $\{0,1\}^n$ with $\mathbf{H}_\infty(I) \ge k$, the distributions $(X, h_X(I))$ and (X,U) are ε-close where X is uniformly random over $\{0,1\}^d$ and U is uniformly random over $\{0,1\}^m$. We say that $\mathcal{H}$ is ρ-universal if for any inputs $I \ne I' \in \{0,1\}^n$ we have $\Pr_{X \xleftarrow{\$} \{0,1\}^d}[h_X(I) \ne h_X(I')] \le \rho$.

Lemma 1 (Leftover-Hash Lemma). *Assume that $\mathcal{H}$ is ρ-universal where $\rho = (1+\alpha)2^{-m}$ for some $\alpha > 0$. Then, for any $k > 0$, it is also a (k, ε)-extractor for $\varepsilon = \frac{1}{2}\sqrt{2^{m-k} + \alpha}$.*

See Theorem 8.37 in [Sho06] for a nicely explained proof of the above lemma.

Pseudorandom Generators. We say that a function $\mathbf{G} : \{0,1\}^m \to \{0,1\}^m$ is a (deterministic) (t, ε)-*pseudorandom generator* (PRG) if $\mathbf{CD}_t(\mathbf{G}(\mathcal{U}_m), \mathcal{U}_n) \leq \varepsilon$.

Game Playing Framework. For our security definitions and proofs we use the code-based game playing framework of [BR06]. 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. In the next section, for all $\mathsf{GAME} \in \{\mathsf{RES}, \mathsf{FWD}, \mathsf{BWD}, \mathsf{ROB}, \mathsf{SROB}\}$, $\mathcal{A}^{\mathsf{GAME}}$ denotes the output of the adversary. We let $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{GAME}} = 2 \times \Pr[\mathsf{GAME}^{\mathcal{A}} \Rightarrow 1] - 1$. 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 PRNG with Input: Modeling and Security

In this section we give formal modeling and security definitions for PRNGs with input.

Definition 1 (PRNG with input). *A PRNG 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:*

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

The integer n is the state length*, ℓ is the output length and p is the input length of $\mathcal{G}$.*

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 PRNG 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 PRNG. In other words, the distribution sampler models potentially adversarial environment (or "nature") where our PRNG is forced to operate. Unlike prior work, we model the distribution sampler *explicitly*, and believe that such modeling is one of the important technical and conceptual contributions of our work.

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:

- σ' is the new state for $\mathcal{D}$.
- $I \in \{0,1\}^p$ is the next input for the $\mathsf{refresh}$ algorithm.
- γ is some *fresh entropy estimation* of I, as discussed below.
- 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:[1]

$$\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_{\mathcal{D}}}, z_1, \ldots, z_{q_{\mathcal{D}}}, \gamma_1, \ldots, \gamma_{q_{\mathcal{D}}}) \geq \gamma_j \tag{4}$$

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 $\sigma_0 = 0$.

We now explain the reason for explicitly requiring $\mathcal{D}$ to output the entropy estimate γ_j used in (4). Most complex PRNGs, including the Linux RNG, are worried about the situation where the system might enter a prolonged state where no new entropy is inserted in the system. Correspondingly, such PRNGs typically include some ad hoc *entropy estimation procedure* E whose goal is to block the PRNG from outputting output value R_j until the state has not 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 [SV03]. This means that if we require our PRNG $\mathcal{G}$ 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, as part of this work we will 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 $\mathsf{refresh}$ and next procedures, meaning that the latter can and *should* be evaluated independently of the "accuracy" of E.

Motivated by these considerations, we do not insist on any "entropy estimation" procedure as a mandatory part of the PRNG design, allowing us to elegantly side-step the practical and theoretical impossibility of sound entropy estimation. Instead, we chose to place the burden of entropy estimations on $\mathcal{D}$ *itself*, which allows us to concentrate on the *provable* security of the $\mathsf{refresh}$ and next procedures. In particular, in our security definition we will not attempt to verify if $\mathcal{D}$'s claims are accurate (as we said, this appears hopeless without some kind of heuristics), but will only require security when $\mathcal{D}$ is *legitimate*, as defined in (4). 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 PRNG operations (which will only use the "input part" 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 $\mathcal{D}$ is "comfortable" with. 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![2] 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[3] crosses some entropy threshold γ^* (since otherwise $\mathcal{D}$ gave us "no reason" to expect any security).

Remark 1. Notice, since in our syntax we did not want to assume that $\mathcal{D}$ knows a bound on the number of calls $q_{\mathcal{D}}$ made to it, we let $\mathcal{D}$ compute the values (I, γ, z) (and update its state σ) one-by-one. This seems to suggest that our attacker $\mathcal{A}$ needs to make j calls $\mathcal{D}$ to eventually learn the "leaked" values γ_j and z_j. However, from a technical point of satisfying the worst-case legitimacy condition (4), we can assume without loss of generality (wlog) that $\mathcal{A}$ learns *all* the $q_{\mathcal{D}}$ values (γ_j, z_j) in the very first leakage z_1. Indeed, in its very first iteration $\mathcal{D}$ could (wlog) compute all $q_{\mathcal{D}}$ iterations, and set the modified first leakage $z'_1 = (\gamma_1, z_1, \ldots, \gamma_{q_{\mathcal{D}}}, z_{q_{\mathcal{D}}})$ (and subsequent $z'_2 = \ldots = z'_{q_{\mathcal{D}}} = \emptyset$) without affecting the bound in (4).

---

[1] Since conditional min-entropy is defined in the worst-case manner in (2), 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] Jumping ahead, setting $\gamma_j = 0$ corresponds to the $\mathsf{bad\text{-}refresh}(I_j)$ oracle in the earlier modeling of [BH05], which is not explicitly provided in our model.

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

3.2 Security Notions

In the literature, four security notions for a PRNG with input have been proposed: *resilience* (RES), *forward security* (FWD), *backward security* (BWD) and *robustness* (ROB), with the latter being the strongest notion among them. We now define the analogs of this notions in our stronger adversarial model, later comparing our modeling with the prior modeling of [BH05]. Each of the games below is parametrized by some parameter γ^* which is part of the claimed PRNG security, and intuitively measures the minimal "fresh" entropy in the system when security should be expected. In particular, minimizing the value of γ^* corresponds to a stronger security guarantee. When γ^* is clear from the context, we omit it for the game description (e.g., write ROB instead of $\mathsf{ROB}(\gamma^*)$).

All four security games ($\mathsf{RES}(\gamma^*)$, $\mathsf{FWD}(\gamma^*)$, $\mathsf{BWD}(\gamma^*)$, $\mathsf{ROB}(\gamma^*)$) are described using the game playing framework discussed earlier, and share the same initialize and finalize procedures in Figure 1 below. As we mentioned, our overall adversary is modeled via a pair of adversaries $(\mathcal{A}, \mathcal{D})$, where $\mathcal{A}$ is the actual attacker and $\mathcal{D}$ is a stateful distribution sampler. We already discussed the distribution sampler $\mathcal{D}$, so we turn to the attacker $\mathcal{A}$, whose goal is to guess the correct value b picked in the initialize procedure, which also returns to $\mathcal{A}$ the public value seed, and initializes several important variables: corruption flag corrupt, "fresh entropy counter" c, state S and sampler's $\mathcal{D}$ initial state σ.[4] In each of the games (RES, FWD, BWD, ROB), $\mathcal{A}$ has access to several oracles depicted in depicted in Figure 2. We briefly discuss these oracles:

proc. initialize $\mathsf{seed} \xleftarrow{\$} \mathsf{setup}$; $\sigma \leftarrow 0$; $S \xleftarrow{\$} \{0,1\}^n$; $c \leftarrow n$; $\mathsf{corrupt} \leftarrow \mathsf{false}$; $b \xleftarrow{\$} \{0,1\}$ 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 $c \geq \gamma^*$,
    $\mathsf{corrupt} \leftarrow \mathsf{false}$
OUTPUT (γ, z)
	proc. next-ror
	IF $\mathsf{corrupt} = \mathsf{true}$,
	    RETURN
	$(S, R_0) \leftarrow \mathsf{next}(S)$
	$R_1 \xleftarrow{\$} \{0,1\}^\ell$
	OUTPUT R_b
		proc. get-next
		$(S, R) \leftarrow \mathsf{next}(S)$
		IF $\mathsf{corrupt} = \mathsf{true}$,
		    $c \leftarrow 0$
		OUTPUT R
			proc. get-state
			$c \leftarrow 0$, $\mathsf{corrupt} \leftarrow \mathsf{true}$
			OUTPUT S
			
			proc. set-state(S^*)
			$c \leftarrow 0$, $\mathsf{corrupt} \leftarrow \mathsf{true}$
			$S \leftarrow S^*$

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

- $\mathcal{D}$-refresh. This is the key procedure where the distribution sampler $\mathcal{D}$ is run, and where its output I is used to refresh the current state S. Additionally, one adds the amount of fresh entropy γ to the entropy counter c, and resets the corrupt flag to false when c crosses the threshold γ^*. The values of γ and the leakage z are also returned to $\mathcal{A}$. We denote by $q_{\mathcal{D}}$ the number of times $\mathcal{A}$ calls $\mathcal{D}$-refresh (and, hence, $\mathcal{D}$), and notice that by our convention (of including oracle calls into run-time calculations) the total run-time of $\mathcal{D}$ is implicitly upper bounded by the run-time of $\mathcal{A}$.
- next-ror/get-next. These procedures provide $\mathcal{A}$ with either the real-or-random challenge (provided that corrupt = false) or the true PRNG output. As a small subtlety, a "premature" call to get-next before

[4] With a slight loss of generality, we assume that when S is random it is safe to set the corruption flag corrupt to false.

$\mathsf{corrupt} = \mathsf{false}$ resets the counter c to 0, since then $\mathcal{A}$ might learn something non-trivial about the (low-entropy) state S in this case.[5] We denote by q_R the total number of calls to next-ror and get-next.

- get-state/set-state. These procedures provide $\mathcal{A}$ with the ability to either learn the current state S, or set it to any value S^*. In either case c is reset to 0 and corrupt is set to true. We denote by q_S the total number of calls to get-state and set-state.

We can now define the corresponding security notions for PRNGs with input. For convenience, in the sequel we sometime denote the "resources" of $\mathcal{A}$ by $T = (t, q_{\mathcal{D}}, q_R, q_S)$.

Definition 2 (Security of PRNG with input). *A pseudo-random number generator with input $\mathcal{G} =$ (setup, refresh, next) is called $(T = (t, q_{\mathcal{D}}, q_R, q_S), \gamma^*, \varepsilon)$-robust (resp. resilient, forward-secure, backward-secure), 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 ROB(γ^*) (resp, RES(γ^*), FWD(γ^*), BWD(γ^*)) is at most ε, where:*

- ROB(γ^*) *is the unrestricted game where $\mathcal{A}$ is allowed to make the above calls.*
- RES(γ^*) *is the restricted game where $\mathcal{A}$ makes no calls to* get-state/set-state *(i.e., $q_S = 0$).*
- FWD(γ^*) *is the restricted game where $\mathcal{A}$ makes no calls to* set-state *and a single call to* get-state *(i.e., $q_S = 1$) which is the very last oracle call $\mathcal{A}$ is allowed to make.*
- BWD(γ^*) *is the restricted game where $\mathcal{A}$ makes no calls to* get-state *and a single call to* set-state *(i.e., $q_S = 1$) which is the very first oracle call $\mathcal{A}$ is allowed to make.*

Intuitively, (a) resilience protects the security of the PRNG when not corrupted against arbitrary distribution samplers $\mathcal{D}$, (b) forward security protects past PRNG outputs in case the state S gets compromised, (c) backward security security ensures that the PRNG can successfully recover from state compromise, provided enough fresh entropy is injected into the system, (d) robustness ensures arbitrary combination of the above. Hence, robustness is the strongest and the resilience is the weakest of the above four notions. In particular, all our provable constructions will satisfy the robustness notion, but we will use the weaker notions to better pinpoint some of our attacks.

3.3 Comparison to Barak-Halevi Model

Barak-Halevi Construction. We briefly recall the elegant construction of PRNG with input due to Barak and Halevi [BH05], since it will help us illustrate the key new elements (and some of the definitional choices) of our new model. This construction (which we call BH) involves a randomness extraction function $\mathsf{Extract} : \{0,1\}^p \longrightarrow \{0,1\}^n$ and a standard deterministic PRG $\mathbf{G} : \{0,1\}^n \longrightarrow \{0,1\}^{n+\ell}$. As we explain below, the modeling of [BH05] did not have an explicit setup algorithm, and the refresh and next algorithms are given below:

- $\mathsf{refresh}(S, I) = \mathbf{G}'(S \oplus \mathsf{Extract}(I))$
- $\mathsf{next}(S) = \mathbf{G}(S)$

Above $\mathbf{G}'$ denotes the truncation of $\mathbf{G}$ to the first n output bits. However, as we explain later, we will also consider the "simplified BH" construction, where $\mathbf{G}'$ is simply the identity function (i.e., $\mathsf{refresh}(S, I) = S \oplus \mathsf{Extract}(I)$).

Entropy Accumulation. Barak and Halevi proved the robustness of this construction in a model very similar to ours (indeed, their model was the inspiration for this work), but with several important differences. The most crucial such difference involves the modeling of the inputs I_j which are fed to the refresh procedure. Unlike our modeling, where the choice of such inputs and *their "fresh entropies"* γ_j is completely left to the

[5] We could slightly strengthen our definition, by only reducing c by ℓ bits in this case, but chose to go for a more conservative notion.

distribution sampler $\mathcal{D}$ (via the $\mathcal{D}$-refresh procedure), the BH modeling only considered the following two extremes of our model. The attacker could either call the good-refresh procedure, which must produce an input I of *fresh entropy* γ *higher than the entropy threshold* γ^*, or call the bad-refresh procedure with an arbitrary, maliciously specified input I^*. Informally, the call to bad-refresh should not compromise the PRNG security whenever the compromised flag corrupt = false, while the call to good-refresh should result in an immediate "recovery", and always resets corrupt = true.

Hence, our key conceptual strengthening of the work of [BH05] will require security even if the entropy is accumulated slowly (and maliciously!), as opposed to in "one shot" (or "delayed" by calls to bad-refresh). Namely, we insist that a good PRNG with input should be able to recover from compromise as long as the *total* amount of fresh entropy accumulated over some *potentially long* period of time crosses the threshold γ^*, instead of insisting that there must be *one* very high-entropy sample to aid the recovery. We informally term this new required property of PRNGs with input (which is very closely related to our formal notion of backward security) *entropy accumulation*, and notice that practical PRNGs, such a the Linux PRNG, seem to place a lot of (heuristic) effort in trying to achieve this property.

Unfortunately, we will show that the *BH construction is not entropy accumulating*, in general. Hence, their construction does not necessarily meet our stronger notion of robustness (or even backward security). Before presenting our attack on the BH construction, though, we discuss some other less critical differences between our models, since they will also help to simplify our presentation of the attack.

Entropy Estimates. Related to the above, [BH05] did not require $\mathcal{D}$ to explicitly output the entropy estimate γ. As we mentioned, though, this was replaced by the implicit requirement that the call to good-refresh must produce an input I with fresh entropy $\gamma \geq \gamma^*$. In contrast, our explicit modeling (justified in detail in Section 3.1) allows us to meaningfully formalize the notion of "entropy accumulation", by keeping a well defined fresh entropy counter c, and resetting corrupt = false when $c \geq \gamma^*$.

Importance of setup. As we mentioned, the modeling of [BH05] did not have an explicit setup algorithm to initialize public parameters seed. Instead, they assumed that the required randomness extractor Extract in their construction is good enough to extract nearly ideal randomness from any high-entropy distribution I output by the good-refresh procedure. Ideally, we would like to make no other assumptions about I except its min-entropy. Unfortunately, it is well known that no deterministic extractor is capable to simultaneously extract good randomness from *all* efficiently samplable high-entropy distributions (e.g., consider nearly full entropy distribution I which is random, except the first bit of Extract(I) is 0). This leaves us with two options. The first option, which seemed to be the preferred choice by [BH05], is to restrict the family of permitted high-entropy distributions I. While following this option is theoretically possible in our model as well, we find it to be somewhat restrictive and cumbersome to define, since we would like to allow our distribution sampler to output "variable-length" high-entropy distributions, where entropy might be accumulated very slowly over time.

Instead, we chose to follow the second option, which is much more universally accepted in the randomness extractor literature [NZ96]: to assume the existence of the setup procedure which will output some public parameters seed which could be used by the procedures next and refresh. Applied to the construction of [BH05], for example, this will allow one to consider a *seeded* extractor Extract inside their refresh procedure, which can now extract entropy from *all* high-entropy distributions (see the resulting definition of seeded (k, ε)-extractors in Section 2). As a warning, this nice extra generality comes at a price that the public parameter seed is not passed to the distribution sampler $\mathcal{D}$, since otherwise $\mathcal{D}$ can still produce high-entropy (but adversarial) samples I such that next(refresh(0^n, I)) always starts with a 0 bit. Although slightly restrictive, this elegantly side-steps the impossibility result above, while accurately modeling many real-life situations, where it is unreasonable to assume that the "nature" $\mathcal{D}$ would somehow bias its samples I depending on some random parameter seed chosen inside the initialization procedure.

State Pseudorandomness. Barak and Halevi [BH05] also insisted that the state S is indistinguishable from random once corrupt = false. While true in their specific construction (analyzed in their weaker model), we think that demanding this property is simultaneously too restrictive and also not very well motivated as the *mandatory* part of a general security *definition*. For example, imagine a PRNG where the state S includes a (never random) Boolean flag which keeps track if the last PRNG call was made to the next procedure. We see a potential efficiency benefit gained by keeping such a flag (e.g., adding this flag will allow us to speed up the subsequent next procedure in our main construction in Section 4), but see no reason why storing such a harmless flag makes such this PRNG design "insecure". In fact, even ignoring the "harmless flag" optimization above, our main construction in Section 4 also will not satisfy this property the very moment $\mathsf{corrupt} = \mathsf{false}$. Instead, we will only make S pseudorandom when the first call to next is made, which will be sufficient to ensure the security of generated bits R anyway.

Indeed, we believe it is better to leave it to the RNG designers to decide on a *particular form* of state pseudorandomness which will aid their security proof, like "ignoring" the harmless flag, or waiting until the first call to next in our construction, etc. For example, in Section 3.4 we will define two simpler, but more specialized notions of RNG security called "preserving" and "recovering" security. Both of these notions will demand a certain *carefully chosen* form of state pseudorandomness (and more!) in a way that, when taken together, will *automatically* imply our notion of robustness. In particular, our main construction will satisfy both of these specialized notions, without satisfying the strict (but not important by itself) state pseudorandomness notion from [BH05].

Interestingly, looking at the analysis of [BH05], the (truncated) PRG $\mathbf{G}'$ inside the $\mathsf{refresh}$ procedure is only needed to ensure the state pseudorandomness of their construction. In other words, if one drops (only the) state pseudorandomness from the BH model, *the "simplified BH" construction is already robust in their model*. Motivated by this, we first give a very strong attack on the simplified BH construction in our stronger model, for *any* extractor $\mathsf{Extract}$ and PRG $\mathbf{G}$. This already illustrates the main difference between our models in terms of entropy accumulation. Then we show a more artificial (but still valid) attack on the "full BH" construction.

Attack on Simplified BH. Consider the following very simple distribution sampler $\mathcal{D}$. At any time period, it simply sets $I = \alpha^p$ (meaning bit α concatenated p times) for a fresh and random bit α, and also sets entropy estimate $\gamma = 1$ and leakage $z = \emptyset$. Clearly, $\mathcal{D}$ is legitimate, as the min-entropy of I is 1, even conditioned on the past and the future. Hence, for any entropy threshold γ^*, the simplified BH construction must regain security after γ^* calls to the $\mathcal{D}$-$\mathsf{refresh}$ procedure following a state compromise. Now consider the following simple attacker $\mathcal{A}$ attacking the backward security (and, thus, robustness) of the simplified BH construction. It calls $\mathsf{set\text{-}state}(0^n)$, and then makes γ^* calls to $\mathcal{D}$-$\mathsf{refresh}$ followed by many calls to $\mathsf{next\text{-}ror}$. Let us denote the value of the state S after j calls to $\mathcal{D}$-$\mathsf{refresh}$ by S_j, and let $Y(0) = \mathsf{Extract}(0^p)$, $Y(1) = \mathsf{Extract}(1^p)$. Then, recalling that $\mathsf{refresh}(S, I) = S \oplus \mathsf{Extract}(I)$ and $S_0 = 0^n$, we see that $S_j = Y(\alpha_1) \oplus \ldots \oplus Y(\alpha_j)$, where $\alpha_1 \ldots \alpha_j$ are random and independent bits. In particular, at any point of time there are only two possible values for S_j: if j is even, then $S_j \in \{0^n, Y(0) \oplus Y(1)\}$, and, if j is odd, then $S_j \in \{Y(0), Y(1)\}$. In other words, despite receiving γ^* random and independent bits from $\mathcal{D}$, the $\mathsf{refresh}$ procedure failed to accumulate more than 1 bit of entropy in the final state $S^* = S_{\gamma^*}$. In particular, after γ^* calls to $\mathcal{D}$-$\mathsf{refresh}$, $\mathcal{A}$ can simply try both possibilities for S^* and easily distinguish real from random outputs with advantage arbitrarily close to 1 (by making enough calls to $\mathsf{next\text{-}ror}$).

This shows that the simplified BH construction is *never* backward secure, despite being robust (modulo state pseudorandomness) in the model of [BH05].

Attack on "Full" BH. The above attack does not immediately extend to the full BH construction, due to the presence of the truncated PRG $\mathbf{G}'$. Instead, we show a less general attack for some (rather than any) extractor $\mathsf{Extract}$ and PRG $\mathbf{G}$. For $\mathsf{Extract}$, we simply take any good extractor (possibly seeded) where $\mathsf{Extract}(0^p) = \mathsf{Extract}(1^p) = 0^n$. Such an extractor exists, since we can take any other initial extractor $\mathsf{Extract}'$, and simply modify it on inputs 0^p and 1^p, as above, without much affecting its extraction properties on *high-entropy* distributions I. By the same argument, we can take any good PRG $\mathbf{G}$ where $\mathbf{G}(0^n) = 0^{n+\ell}$, which means that $\mathbf{G}'(0^n) = 0^n$.

With these (valid but artificial) choices of $\mathsf{Extract}$ and $\mathbf{G}$, we can keep the same distribution sampler $\mathcal{D}$ and the attacker $\mathcal{A}$ as in the simplified BH example. Now, however, we observe that the state S always remains equal to 0^n, irrespective of whether is it updated with $I = 0^p$ or $I = 1^p$, since the new state $S' = \mathbf{G}'(S \oplus \mathsf{Extract}(I)) = \mathbf{G}'(0^n \oplus 0^n) = 0^n = S$. In other words, we have not gained even a single bit of entropy into S, which clearly breaks backward security in this case as well!

One may wonder if we can have a less obvious attack for any $\mathsf{Extract}$ and $\mathbf{G}$, much like in the simplified BH case. This turns out to be an interesting and rather non-trivial question. Indeed, the value of the state S_j after j calls to $\mathcal{D}$-$\mathsf{refresh}$ with inputs $I_1 \ldots I_j$ is equal to the "CBC-MAC" computation, with input $\boldsymbol{Y} = (Y_1 \ldots Y_j)$ and the initial value S_0, where $Y_j = \mathsf{Extract}(I_j)$:

$$S_j = \mathbf{G}'(Y_j \oplus \mathbf{G}'(Y_{j-1} \ldots \oplus \mathbf{G}'(Y_1 \oplus S_0) \ldots))$$

Moreover, we only care about the case when $\mathbf{H}_\infty(\boldsymbol{I}) \geq \gamma^*$, which, under appropriate assumptions on $\mathsf{Extract}$, would translate to a high-entropy guarantee on $\boldsymbol{Y}$. In this case, it is tempting to use the work of [DGH+04], who showed that the CBC-MAC is a good randomness extractor on high-entropy inputs $\boldsymbol{Y}$, provided that the truncated PRG $\mathbf{G}'$ is modeled as a random *permutation*. This result gives us hope that the full BH construction might be secure in our model, possibly under strong enough assumptions on the PRG $\mathbf{G}$ and/or the extractor $\mathsf{Extract}$. Unfortunately, aside from assuming that $\mathbf{G}'$ is (close to) a random permutation, we cannot directly use the results of [DGH+04], since the initial state S_0 could be set by $\mathcal{A}$ in a way correlated with the inputs Y_j, as well as the "block cipher" $\mathbf{G}'$ (which invalidates the analysis of [DGH+04]).

Instead of following this interesting, but rather speculative direction, in Section 4 we give an almost equally simple construction which is *provably robust* in the standard model, without any idealized assumptions.

3.4 Simpler Notions of PRNG Security

We define two properties of a PRNG with input which are intuitively simpler to analyze than the full robustness security. We show that these two properties, taken together, imply robustness.

Recovering Security. We define a notion of *recovering security*. It considers an attacker that compromises the state to some arbitrary value S_0. Following that, sufficiently many $\mathcal{D}$-$\mathsf{refresh}$ calls with sufficient entropy are made so as to set the $\mathsf{corrupt}$ flag to false and resulting in some updated state S. Then the output $(S^*, R) \leftarrow \mathsf{next}(S)$ looks indistinguishable from uniform. The formal definition is slightly more complicated since the attacker also gets to adaptively choose *when* to start using $\mathcal{D}$-$\mathsf{refresh}$ calls to update the state. Formally, we consider the following security game with an attacker $\mathcal{A}$, a sampler $\mathcal{D}$, and bounds $q_{\mathcal{D}}, \gamma^*$.

- The challenge chooses a seed $\mathsf{seed} \xleftarrow{\$} \mathsf{setup}$, and a bit $b \xleftarrow{\$} \{0,1\}$ 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 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 $S_0 \in \{0,1\}^n$ and an integer d such that $k + d \leq q_{\mathcal{D}}$ and $\sum_{j=k+1}^{k+d} \gamma_j \geq \gamma^*$.
- For $j = 1, \ldots, d$, the challenger computes

$$S_j := \mathsf{refresh}(S_{j-1}, I_{k+j}, \mathsf{seed}).$$

  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^*.

We define the advantage of the attacker $\mathcal{A}$ and sampler $\mathcal{D}$ in the above game as $|2\Pr[b^* = b] - 1|$.

Definition 3 (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*.*

Preserving Security. We define a simple notion of *preserving security.* Intuitively, it says that if the state S_0 starts uniformly random and uncompromised, and then is 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.

Formally, we consider the following security game with an attacker $\mathcal{A}$.

- 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.
- $\mathcal{A}$ gets seed and specifies 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 $b = 0$, $\mathcal{A}$ is given $(S^*, R) = \mathsf{next}(S_d)$ and if $b = 1$, $\mathcal{A}$ is given $(S^*, R) \leftarrow \{0,1\}^{n+\ell}$.
- $\mathcal{A}$ outputs a bit b^*.

We define the advantage of the attacker $\mathcal{A}$ in the above game as $|2\Pr[b^* = b] - 1|$.

Definition 4 (Preserving Security). *A PRNG with input has* (t, ε)-preserving security *if the advantage of any attacker* $\mathcal{A}$ *running in time* t *in the above game is at most* ε*.*

We now show that, taken together, recovering and preserving security notions imply the full notion of strong robustness.

Theorem 1. *If a PRNG with input has both* $(t, q_{\mathcal{D}}, \gamma^*, \varepsilon_r)$*-recovering security and* (t, ε_p)*-preserving security, then it is* $((t', q_{\mathcal{D}}, q_R, q_S), \gamma^*, q_R(\varepsilon_r + \varepsilon_p))$*-robust where* $t' \approx t$*.*

3.5 Proof of Theorem 1

We will refer to the attacker's queries to either the $\mathsf{get\text{-}next}$ or $\mathsf{next\text{-}ror}$ oracle in the robustness game as "next queries". We assume that the attacker makes exactly q_R of them. We say that a next query is *uncompromised* if $\mathsf{corrupt} = \mathsf{false}$ during the query, and we say it is *compromised* otherwise. Without loss of generality, we will assume that all compromised next queries that the attacker makes are to $\mathsf{get\text{-}next}$ and *not* $\mathsf{next\text{-}ror}$ (since $\mathsf{next\text{-}ror}$ does not do/output anything when $\mathsf{corrupt} = \mathsf{true}$).

We partition the uncompromised next queries into two subcategories: *preserving* and *recovering.* We say that an uncompromised next query is *preserving* if the $\mathsf{corrupt}$ flag remained set to false throughout the entire period between the previous next query (if there is one) and the current one. Otherwise, we say that an uncompromised next query is *recovering.* With any recovering next query, we can associate a corresponding *most recent entropy drain (mRED)* query which is the most recent query to either $\mathsf{get\text{-}state}$, $\mathsf{set\text{-}state}$, $\mathsf{get\text{-}next}$ that precedes the current next query. An mRED query must set the cumulative entropy estimate to $c = 0$. Moreover, with any recovering next query, we associate a corresponding sequence of *recovering samples* $\bar{I} = (I_j, \ldots, I_{j+d-1})$ which are output by all the calls to the $\mathcal{D}$-$\mathsf{refresh}$ oracle that precede the recovering next query, but follow the associated mRED query. It is easy to see that any such sequence of recovering samples $\bar{I}$ must satisfy the entropy requirements $\sum_{i=j}^{j+d-1} \gamma_i \geq \gamma^*$ where the ith call to $\mathcal{D}$-$\mathsf{refresh}$ oracle outputs (I_i, γ_i, z_i).

We define several hybrid games. Let **Game** 0 be the real-or-random security game as defined in Figure 2. Let **Game** i be a modification of this game where, for the first i next queries, if the query is *uncompromised*, then the challenger *always* chooses $(S, R) \leftarrow \{0,1\}^{\ell+n}$ uniformly at random during the query rather than using the $\mathsf{next}()$ function. As an intermediate, we also define a hybrid Game $(i + \frac{1}{2})$, which lies between

Game i and **Game** $(i+1)$. In particular, in **Game** $(i+\frac{1}{2})$, if the $(i+1)$st next query is *preserving* than the challenger acts as in **Game**$(i+1)$ and chooses a random S, R, and otherwise it acts as in **Game** i and follows the original oracle specification. In all of these games, the output of the game is the output of the finalize oracle at the end, which is 1 if the attacker correctly guesses the challenge bit, and 0 otherwise.

We claim that for all $i \in \{0, \ldots, q_R - 1\}$, **Game** i is indistinguishable from **Game** $(i+\frac{1}{2})$, that in turn is indistinguishable from **Game** $(i+1)$.

Claim. Assuming that the PRNG has (t, ε_p)-preserving security, then for any attacker/distinguisher $\mathcal{A}, \mathcal{D}$ running in time $t' \approx t$, we have $|\Pr[(\textbf{Game } i) = 1] - \Pr[(\textbf{Game } i + \frac{1}{2}) = 1]| \leq \varepsilon_p$.

Proof. Fix attacker/sampler pair $\mathcal{A}, \mathcal{D}$ running in time t'. Note that the two games above only differ in the special case where the $(i+1)$st next query made by $\mathcal{A}$ is *preserving*. Therefore, we can assume w.l.o.g. that $\mathcal{A}$ ensures this is always the case, as it can only maximize advantage.

We define an attacker $\mathcal{A}'$ that has advantage ε_p in the preserving security game. The attacker $\mathcal{A}'$ gets a value seed from its challenger and passes it to $\mathcal{A}$. Then $\mathcal{A}'$ begins running $\mathcal{A}$ and simulating all of the oracles in the robustness security game. It chooses a random "challenge bit" $b \stackrel{\$}{\leftarrow} \{0,1\}$. It simulates all oracle calls made by $\mathcal{A}$ until the $(i+1)$st next query as in **Game** i. In particular, it simulates calls to $\mathcal{D}$-refresh using the code of the sampler $\mathcal{D}$ and updating its state. Note that $\mathcal{A}'$ has complete knowledge of the sampler state σ and the PRNG state S at all times. During the $(i+1)$st next query made by $\mathcal{A}$, the attacker $\mathcal{A}'$ takes all the samples $I_1, \ldots, I_d$ which were output by $\mathcal{D}$ in between the ith and $(1+1)$st next query and gives these to its challenger. It gets back a value (S^*, R_0). If the $(i+1)$st next query made by $\mathcal{A}$ is next-ror the attacker $\mathcal{A}'$ also chooses $R_1 \stackrel{\$}{\leftarrow} \{0,1\}^\ell$ and gives R_b to $\mathcal{A}$ where b is challenge bit randomly picked by $\mathcal{A}'$. In either case, $\mathcal{A}'$ sets the new PRNG state to S^* and continues running the game, simulating all future oracle calls made by $\mathcal{A}$ as in **Game** i. Finally, if $\mathcal{A}$ outputs the bit b^*, the attacker $\mathcal{A}'$ outputs the bit $\tilde{b}^*$ which is set to 1 iff $b = b^*$.

Notice that if the challenge bit of the challenger for $\mathcal{A}'$ is $\tilde{b} = 0$ then this exactly simulates **Game** i for $\mathcal{A}$ and if the challenge bit is $\tilde{b} = 1$ then this exactly simulates **Game** $i+1$. In particular, we can think of the state immediately following the ith next query as being the challenger's randomly chosen value $S_0 \stackrel{\$}{\leftarrow} \{0,1\}^n$, the state immediately preceding the $(i+1)$st next query being S_d which refreshes S_0 with the samples $I_1, \ldots, I_d$, and the state immediately following the query as being either $(S^*, R_0) \leftarrow \text{next}(S_d)$ when $\tilde{b} = 0$ (as in **Game** i) or $(S^*, R_0) \stackrel{\$}{\leftarrow} \{0,1\}^{n+\ell}$ when $b = 1$ (as in **Game** $i+1$). Therefore we have

$$\begin{aligned} \left| \Pr\left[(\textbf{Game } i + 1/2) = 1\right] - \Pr\left[(\textbf{Game } i) = 1\right] \right| &= \left| \Pr[b = b^* | \tilde{b} = 0] - \Pr[b = b^* \mid \tilde{b} = 1] \right| \\ &= \left| 2\Pr[\tilde{b}^* = \tilde{b}] - 1 \right| \\ &\leq \varepsilon_p. \end{aligned}$$

□

Claim. If the PRNG is $(t, q_{\mathcal{D}}, \gamma^*, \varepsilon_r)$-recovering secure, then for any attacker/distinguisher $\mathcal{A}, \mathcal{D}$ running in time $t' \approx t$, we have $|\Pr[(\textbf{Game } i + \frac{1}{2}) = 1] - \Pr[(\textbf{Game } i+1) = 1]| \leq \varepsilon_r$.

Proof. Fix attacker/sampler pair $\mathcal{A}, \mathcal{D}$ running in time t'. Note that the two games above only differ in the special case where the $(i+1)$st next query made by $\mathcal{A}$ is *recovering*. Therefore, we can assume w.l.o.g. that $\mathcal{A}$ ensures this is always the case, as it can only maximize advantage.

We define an attacker $\mathcal{A}'$ such that $\mathcal{A}', \mathcal{D}$ has advantage ε_r in the recovering security game. The attacker $\mathcal{A}'$ gets a value seed from its challenger and passes it to $\mathcal{A}$. Then $\mathcal{A}'$ begins running $\mathcal{A}$ and simulating all of the oracles in the robustness security game. In particular, it chooses a random "challenge bit" $b \stackrel{\$}{\leftarrow} \{0,1\}$ and state $S \stackrel{\$}{\leftarrow} \{0,1\}^n$. It simulates all oracle calls made by $\mathcal{A}$ until right prior to the $(i+1)$st next query as in **Game** i. To simulate calls to $\mathcal{D}$-$\mathsf{refresh}$, the attacker $\mathcal{A}'$ outputs the values γ_k, z_k that it got from its challenger in the beginning, but does not immediately update the current state S. Whenever $\mathcal{A}$ makes an oracle call to $\mathsf{get\text{-}state}, \mathsf{get\text{-}next}, \mathsf{next\text{-}ror}, \mathsf{set\text{-}state}$, $\mathcal{A}$' first makes sufficiently many calls to its $\mathsf{get\text{-}refresh}$ oracle so as to get the corresponding samples I_k that should have been sampled by these prior $\mathcal{D}$-$\mathsf{refresh}$ calls, and refreshes its state S accordingly before processing the current oracle call. When $\mathcal{A}$ makes its $(i+1)$st next query, the attacker $\mathcal{A}'$ looks back and finds the *most recent entropy drain (mRED)* query that $\mathcal{A}$ made, and sets S_0 to the state of the PRNG immediately following that query. Assume $\mathcal{A}$ made d calls to $\mathcal{D}$-$\mathsf{refresh}$ between the mRED query and the $i+1$st next query (these are the "recovering samples"). Then $\mathcal{A}$' gives (S_0, d) to its challenger and gets back (S^*, R_0) and $I_{k+d+1}, \ldots, I_{q_\mathcal{D}}$. It chooses $R_1 \xleftarrow{\$} \{0,1\}^\ell$. If the $(i+1)$st next query made by $\mathcal{A}$ is $\mathsf{next\text{-}ror}$ the attacker $\mathcal{A}'$ also chooses $R_1 \xleftarrow{\$} \{0,1\}^\ell$ and gives R_b to $\mathcal{A}$, where b is challenge bit randomly picked by $\mathcal{A}'$ in the beginning. In either case, $\mathcal{A}'$ sets the new PRNG state to S^* and continues running the game, simulating all future oracle calls made by $\mathcal{A}$ as in **Game** $i+1$ using the values $I_{k+d+1}, \ldots, I_{q_\mathcal{D}}$ to simulate $\mathcal{D}$-$\mathsf{refresh}$ calls. Finally, if $\mathcal{A}$ outputs the bit b^*, the attacker $\mathcal{A}'$ outputs the bit $\tilde{b}^*$ which is set to 1 iff $b = b^*$.

Notice that if the challenge bit of the challenger for $\mathcal{A}'$ is $\tilde{b} = 0$ then this exactly simulates **Game** $i + 1/2$ for $\mathcal{A}$ and if the challenge bit is $\tilde{b} = 1$ then this exactly simulates **Game** $i+1$. In particular, we can think of the state immediately immediately following the mRED query as S_0 and the state immediately preceding the $(i+1)$st next query being S_d which refreshes S_0 with the samples $I_{k+1}, \ldots, I_{k+d}$, and the state immediately following the query as being either $(S^*, R_0) \leftarrow \mathsf{next}(S_d)$ when $\tilde{b} = 0$ (as in **Game** $i+1/2$) or $(S^*, R_0) \xleftarrow{\$} \{0,1\}^{n+\ell}$ when $b = 1$ (as in **Game** $i+1$). Also, we note that $\mathcal{A}'$ is a valid attacker since the recovering samples must satisfy $\sum_{j=k+1}^{k+d} \gamma_j \geq \gamma^*$ if the $(i+1)$st next query is recovering. Therefore we have:

$$\begin{aligned} \left| \Pr\left[(\textbf{Game } i+1/2) = 1\right] - \Pr\left[(\textbf{Game } i) = 1\right] \right| &= \left| \Pr[b = b^* | \tilde{b} = 0] - \Pr[b = b^* \mid \tilde{b} = 1] \right| \\ &= \left| 2\Pr[\tilde{b}^* = \tilde{b}] - 1 \right| \\ &\leq \varepsilon_r. \end{aligned}$$

□

Combining the above two claims, and using the hybrid argument, we get:

$$|\Pr[(\textbf{Game } 0) = 1] - \Pr[(\textbf{Game } q_R) = 1]| \leq q_R(\varepsilon_r + \varepsilon_p).$$

Moreover **Game** q_R is completely independent of the challenger bit b. In particular, all $\mathsf{next\text{-}ror}$ queries return a random $R \xleftarrow{\$} \{0,1\}^\ell$ independent of the challenge bit b. Therefore, we have $\Pr[(\textbf{Game } q_R) = 1] = \frac{1}{2}$. Combining with the above, we see that the attacker's advantage in the original robustness game is $\left|\Pr[(\textbf{Game } 0) = 1] - \frac{1}{2}\right| \leq q_R(\varepsilon_r + \varepsilon_p)$.

4 Provably Secure Construction

Let $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ be a (deterministic) pseudorandom generator where $m < n$. We use the notation $[y]_1^m$ to denote the first m bits of $y \in \{0,1\}^n$. Our construction of PRNG 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 $(S', R) = \mathbf{G}(U)$.

Notice that we are assuming each input I is in $\{0,1\}^n$. This is without loss of generality: we can take shorter inputs and pad them with 0s, or take longer inputs and break them up into n-bit chunks, calling the $\mathsf{refresh}$ procedure iteratively on each chunk.

On-line Extractor. Let's look at what happens if we start in some state S and call the $\mathsf{refresh}$ procedure d-times with the samples $I_{d-1}, \dots, I_0$ (it will be convenient to index these in reverse order). Then the new state at the end of this process will be

$$S' := S \cdot X^d + I_{d-1} \cdot X^{d-1} + \dots + I_1 \cdot X + I_0.$$

Let $\bar{I} := (I_{d-1}, \dots, I_0)$ be the concatenation of all d samples. In the analysis we rely on the fact that the *polynomial evaluation* hash function defined by $h_X(\bar{I}) := \sum_{j=0}^{d-1} I_j \cdot X^j$ is $(d/2^n)$-universal meaning that the probability of any two distinct inputs colliding is at most $d/2^n$ over the random choice of X. In particular, we can think of our refresh procedure as computing this hash function in an *on-line* manner, processing the inputs I_j one-by-one without knowing the total number of future samples d, and keeping only a short local state.[6] In particular, the updated state after the d refreshes is $S' = S \cdot X^d + h_X(\bar{I})$. Unfortunately, $h_X(\cdot)$ is not sufficiently universal to make it a good extractor, and therefore we cannot argue that S' itself is random as long as $\bar{I}$ has entropy. Therefore, we need to apply an additional hash function $h'_{X'}(Y) = [X' \cdot Y]_1^m$ which takes as input $Y \in \{0,1\}^n$ and outputs a value $h_{X'}(Y) \in \{0,1\}^m$. We show that the composition function $h^*_{X,X'}(\bar{I}) = h'_{X'}(h_X(\bar{I}))$ is a good randomness extractor. Therefore, during the evaluation of $(S'', R) = \mathsf{next}(S')$, the value

$$U = [X' \cdot S']_1^m = [X' \cdot S \cdot X^d]_1^m + h^*_{X,X'}(\bar{I})$$

is uniformly random as long as the refreshes $\bar{I}$ jointly have sufficient entropy. This is the main idea behind our construction. We formalize this via the following lemma, which provides the key to proving our main theorem.

Lemma 2. *Let d, n, m be integers, let $X, X', Y \in \mathbb{F}_{2^n}$, $\bar{I} = (I_{d-1}, \dots, I_0) \in \mathbb{F}_{2^n}^d$. Define the hash function families:*

$$h_X(\bar{I}) := \sum_{j=0}^{d-1} I_j \cdot X^j \quad , \quad h'_{X'}(Y) := [X' \cdot Y]_1^m.$$

$$h^*_{X,X'}(\bar{I}) := h'_{X'}(h_X(\bar{I})) = \left[X' \cdot \sum_{j=0}^{d-1} I_j \cdot X^j \right]_1^m.$$

Then the hash-family $\mathcal{H} = \{h^*_{X,X'}\}$ is $2^{-m}(1 + d \cdot 2^{m-n})$-universal. In particular it is a (k, ε)-extractor as long as:

$$k \geq m + 2\log(1/\varepsilon) + 1 \quad , \quad n \geq m + 2\log(1/\varepsilon) + \log(d) + 1.$$

Proof. For the first part of the lemma, fix any

$$\bar{I} = (I_{d-1}, \dots, I_0) \neq \bar{I}' = (I'_{d-1}, \dots, I'_0).$$

[6] The fact that polynomial evaluation can be computed in such on-line manner is called Horner's method. It has countless applications in algorithm design and many areas of computer science.

Then:

$$\Pr_{X,X'}[h^*_{X,X'}(\bar{I}) = h_{X,X'}(\bar{I}')] \leq \Pr_X[h_X(\bar{I}) = h_X(\bar{I}')] + \Pr_{X,X'}\left[h_{X'}(Y) = h_{X'}(Y') \;\middle|\; \begin{array}{c} Y \neq Y' \\ Y := h_X(\bar{I}), \\ Y' := h_X(\bar{I}') \end{array}\right]$$
$$\leq \Pr_X\left[\sum_{j=0}^{d-1}(I_j - I'_j)\cdot X^j = 0\right] + 2^{-m}$$
$$\leq d/2^n + 2^{-m} = 2^{-m}(1 + d2^{m-n}).$$

For proving the second part, we use the fact that $h_{X,X'}$ is $2^{-m}(1+\alpha)$-universal for $\alpha = d \cdot 2^{m-n}$. Hence, it is also a (k,ε)-extractor where $\varepsilon \leq \sqrt{2^{m-k}+\alpha} = \sqrt{2^{m-k}+d2^{m-n}}$ (See Lemma 1). This is ensured by our parameter choice. □

The proof of Lemma 2 will be crucially used in establishing our main theorem below.

Theorem 2. *Let $n > m, \ell, \gamma^*$ be integers. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a deterministic (t, ε_{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 PRNG with input where $t' \approx t$, $\varepsilon = q_R(2\varepsilon_{prg} + q_{\mathcal{D}}^2\varepsilon_{ext} + 2^{-n+1})$ as long as $\gamma^* \geq m + 2\log(1/\varepsilon_{ext}) + 1$, $n \geq m + 2\log(1/\varepsilon_{ext}) + \log(q_{\mathcal{D}}) + 1$.*

We present the proof in the next Section, but now make a few comments. First, it is instructive to split the security bound on ε into two parts (ignoring the "truly negligible" term $q_R \cdot 2^{-n+1}$): "computational" part $\varepsilon_{comp} = 2q_R \cdot \varepsilon_{prg}$ and "statistical" part $\varepsilon_{stat} = q_R q_{\mathcal{D}}^2 \cdot \varepsilon_{ext}$, so that $\varepsilon \approx \varepsilon_{comp} + \varepsilon_{stat}$. Notice, the computational term ε_{comp} is already present in any "input-free" PRNG (or "stream cipher"), where the state S is assumed to never be compromised (so there is no $\mathsf{refresh}$ operation) and $\mathsf{next}(S) = \mathbf{G}(S)$. Also, such stream cipher has state length $n = m$. Thus, we can view the statistical term $\varepsilon_{stat} = q_R q_{\mathcal{D}}^2 \cdot \varepsilon_{ext}$ and the "state overhead" $n - m = 2\log(1/\varepsilon_{ext}) + \log(q_{\mathcal{D}}) + 1$ as the "price" one has to pay to additionally recover from occasional compromise (using fresh entropy gathered by the system).

Second, to slightly reduce the number of parameters in Theorem 2, we can let k be our "security parameter" and set $q_{\mathcal{D}} = q_R = q_S = 2^k$ and $\varepsilon_{ext} = 2^{-4k}$. Then we see that $\varepsilon_{stat} = 2^{3k} \cdot 2^{-4k} = 2^{-k}$, $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$ and we can set $n = m + 2\log(1/\varepsilon_{ext}) + \log(q_{\mathcal{D}}) + 1 = m + 9k + 1$ and $\gamma^* = m + 2\log(1/\varepsilon_{ext}) + 1 = m + 8k + 1$. Summarizing all of these, we get the following Corollary.

Corollary 1. *Let k, m, ℓ, n be integers, where $n \geq m + 9k + 1$. Assume that $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a deterministic (t, ε_{prg})-pseudorandom generator. Then $\mathcal{G}$ is a $((t', 2^k, 2^k, 2^k), m+8k+1, 2^{k+1} \cdot \varepsilon_{prg} + 2^{-k})$-robust PRNG with input, having n-bit state and ℓ-bit output, where $t' \approx t$.*

Coming back to our comparison with the stream ciphers (or "input-free" PRNGs), we see that we can achieve statistical security overhead $\varepsilon_{stat} = 2^{-k}$ (with $q_{\mathcal{D}} = q_R = q_S = 2^k$) at the price of state overhead $n - m = 9k + 1$ (and where entropy threshold $\gamma^* = m + 8k + 1 = n - k$).

Practical Efficiency Optimization. Notice, our current next operation consists of the "randomness extraction" step $U = [X' \cdot S]_1^m$ followed by the "PRG step" $(S', R) = \mathbf{G}(U)$. As explained above, the extraction step is needed to ensure that $\mathbf{G}$ is applied to a statistically random value U. However, after the first call to next is completed, the entire state S becomes pseudorandom (assuming, of course, that $\mathsf{corrupt} = \mathsf{false}$). Hence, after the first call to next there is no need to apply the extraction step again, and we can simply set $U = [S]_1^m$. Namely, if the we call next "prematurely" (i.e., $\mathsf{corrupt} = \mathsf{true}$), then we anyway reset the counter $c = 0$ and start accumulating entropy "from scratch", and, otherwise, the first m bits of S become pseudorandom anyway, so the extraction step becomes redundant from now on.

Moreover, for subsequent calls we do not even need out PRG $\mathbf{G}$ to stretch from m to $n+\ell$ bits, but can use some (possibly faster) PRG $\mathbf{G}'$ from m bits to $m+\ell$ bits.[7] Indeed, we can let $(U,R) = \mathbf{G}'([S]_1^m)$ and set $[S]_1^m = U$, which is all we need to ensure subsequent calls to next use a pseudorandom seed value.

Summarizing the above discussion, in the optimized version we can add a Boolean flag last to our state, which is set to true (only) if the last call to $\mathcal{G}$ was next. When next is called then, if $\mathsf{last} = \mathsf{false}$, we implement the same extraction and PRG step as above: $U = [X' \cdot S]_1^m$, $(S', R) = \mathbf{G}(U)$. However, if $\mathsf{last} = \mathsf{true}$, we let $(U,R) = \mathbf{G}'([S]_1^m)$ and set $[S]_1^m = U$. (Either way, we set $\mathsf{last} = \mathsf{true}$.)

Aside from noticeably improved efficiency, this version has the advantage of having the same complexity of next (after the first call) as traditional stream ciphers (i.e., "input-free PRNGs"). Thus, we add the ability to recover from compromise without any effect on the efficiency of the actual random number generation!

4.1 Proof of Theorem 2

We show that $\mathcal{G}$ satisfies $(t', q_{\mathcal{D}}, \gamma^*, (\varepsilon_{prg} + q_{\mathcal{D}}^2 \varepsilon_{ext}))$-recovering security and $(t', (\varepsilon_{prg} + 2^{-n+1}))$-preserving security. Theorem 2 then follows directly from Theorem 1.

Claim. The PRNG $\mathcal{G}$ has $(t', \varepsilon_{prg} + 2^{-n+1})$-preserving security.

Proof. Let **Game** 0 be the original preserving security game: the game outputs a bit which is set to 1 iff the attacker guesses the challenge bit $b^* = b$. If the initial state is $S_0 \xleftarrow{\$} \{0,1\}^n$, the seed is $\mathsf{seed} = (X, X')$, and the adversarial samples are $I_{d-1}, \ldots, I_0$ (indexed in reverse order where I_{d-1} is the earliest sample) then the refreshed state that incorporates these samples will be $S_d := S_0 \cdot X^d + \sum_{j=0}^{d-1} I_j \cdot X^j$. As long as $X \neq 0$, the value S_d is uniformly random (over the choice of S_0). We consider a modified **Game** 1, where the challenger simply chooses $S_d \xleftarrow{\$} \{0,1\}^n$ and we have

$$|\Pr[(\textbf{Game } 0) = 1] - \Pr[(\textbf{Game } 1) = 1]| \leq 2^{-n}.$$

Let $U = [S_d \cdot X']_1^m$ be the value computed by the challenger during the computation $(S,R) \leftarrow \mathsf{next}(S_d)$ when the challenge bit is $b = 0$. Then, as long as $X' \neq 0$, the value U is uniformly random (over the choice S_d). Therefore, we can define **Game** 2 where the challenger choose $U \xleftarrow{\$} \{0,1\}^n$ during this computation and we have:

$$|\Pr[(\textbf{Game } 1) = 1] - \Pr[(\textbf{Game } 2) = 1]| \leq 2^{-n}.$$

Finally $(S,R) = \mathsf{next}(S_d, \mathsf{seed}) = \mathbf{G}(U)$. Then (S,R) is (t, ε_{prg}) indistinguishable from uniform. Therefore we can consider a modified **Game** 3 where the challenger just choosing (S,R) at random even when the challenge bit is $b = 0$. Since the attacker runs in time $t' \approx t$, we have:

$$|\Pr[(\textbf{Game } 3) = 1] - \Pr[(\textbf{Game } 2) = 1]| \leq \varepsilon_{prg}.$$

Since **Game** 3 is independent of the challenge bit b, we have $\Pr[(\textbf{Game } 3) = 1] = \frac{1}{2}$ and therefore $|\Pr[(\textbf{Game } 0) = 1] - \frac{1}{2}| \leq \varepsilon_{prg} + 2^{-n+1}$. □

Claim. The PRNG $\mathcal{G}$ has $(t', q_{\mathcal{D}}, \gamma^*, (\varepsilon_{prg} + q_{\mathcal{D}}^2 \varepsilon_{ext}))$-recovering security.

Proof. Let **Game** 0 be the original recovering security game: the game outputs a bit which is set to 1 iff the attacker guesses the challenge bit $b^* = b$. We define **Game** 1 where, during the challenger's computation of $(S^*, R) \leftarrow \mathsf{next}(S_d)$ for the challenge bit $b = 0$, it chooses $U \xleftarrow{\$} \{0,1\}^m$ uniformly at random rather than setting $U := [X' \cdot S_d]_1^m$. We argue that

$$|\Pr[(\textbf{Game } 0) = 1] - \Pr[(\textbf{Game } 1) = 1]| \leq q_{\mathcal{D}}^2 \varepsilon_{ext}.$$

[7] For example, if $\mathbf{G}$ is implemented by an ℓ-bit block cipher in the counter mode, we need $1+n/\ell$ block cipher calls to implement $\mathbf{G}$, and only $1 + m/\ell$ calls to implement $\mathbf{G}'$. Hence, we save about $9k/\ell$ calls when using parameters of Corollary 1.

The loss of $q_{\mathcal{D}}^2$ comes from the fact that the attacker can choose the index k and the value d adaptively depending on the seed. In particular, assume that the above does not hold. Then there must exist some values $k^*, d^* \in [q_{\mathcal{D}}]$ such that the above distance is greater than ε_{ext} conditioned on the attacker making exactly k^* calls to get-refresh and choosing d^* refreshes in the game. We show that this leads to a contradiction. Fix the distribution on the subset of samples $\bar{I} = (I_{k^*+1}, \ldots, I_{k+d^*})$ output by $\mathcal{D}$ during the first step of the game, which must satisfy

$$\mathbf{H}_\infty(\bar{I} \mid \gamma_1, \ldots, \gamma_{q_{\mathcal{D}}}, z_1, \ldots, z_{q_{\mathcal{D}}}) \geq \gamma^*.$$

By Lemma 2, the function $h_{X,X'}(\bar{I})$ is a $(\gamma^*, \varepsilon_{ext})$-extractor, meaning that $(X, X', h_{X,X'}(\bar{I}))$ is ε_{ext}-close to (X, X', Z) where Z is random an independent of X, X'. Then, for any fixed choice of k^*, d^*, the way we compute U in **Game** 0:

$$U := [X' \cdot S_d]_1^m = [X' \cdot S_0 X^d]_1^m + h_{X,X'}(\bar{I})$$

is ε_{exst} close to a uniformly random U as chosen in **Game** 1. This leads to a contradiction, showing that the equation holds.

Finally, we define **Game** 2 where, during the challenger's computation of $(S^*, R) \leftarrow \mathsf{next}(S_d)$ for the challenge bit $b = 0$, it chooses (S^*, R) uniformly at random instead of $(S^*, R) \leftarrow \mathbf{G}(U)$ as in **Game** 1. Since the attacker runs in time $t' \approx t$, we have:

$$|\Pr[(\mathbf{Game}\ 2) = 1] - \Pr[(\mathbf{Game}\ 1) = 1]| \leq \varepsilon_{prg}.$$

Since **Game** 2 is independent of the challenge bit b, we have $\Pr[(\mathbf{Game}\ 2) = 1] = \frac{1}{2}$ and therefore $|\Pr[(\mathbf{Game}\ 0) = 1] - \frac{1}{2}| \leq \varepsilon_{prg}$. □

5 Analysis of the Linux PRNGs

The Linux operating system contains two PRNGs with input, `/dev/random` and `/dev/urandom`. They are part of the kernel and used in the OS security services or some cryptographic libraries. We give a precise description[8] of them in our model as a triple $\mathsf{LINUX} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ and we prove the following theorem:

Theorem 3. *The Linux PRNGs `/dev/random` and `/dev/urandom` are not robust.*

Since the actual generator LINUX does not define any seed (*i.e.* the algorithm setup always output the empty string), as mentioned above, it cannot achieve the notion of robustness. However, in Sections 5.8 and 5.8, we additionally mount concrete attacks that would work even if LINUX had used a seed in the underlying hash function or mixing function. The attacks exploit two independent weaknesses, in the entropy estimator and the mixing functions, which would need both to be fixed in order to expect the PRNGs to be secure.

5.1 General Overview

Security Parameters. The LINUX PRNG uses parameters $n = 6144$, $\ell = 80$, $p = 96$. The parameter n can be modified (but requires kernel compilation), and the parameters ℓ (size of the output) and p (size of the input) are fixed. The PRNG outputs the requested random numbers by blocks of $\ell = 80$ bits and truncates the last block if needed.

Internal State. The internal state of LINUX PRNG is a triple $S = (S_i, S_u, S_r)$ where $|S_i| = 4096$ bits, $|S_u| = 1024$ bits and $|S_r| = 1024$ bits. New data is collected in S_i, which is named the *input pool*. Output is generated from S_u and S_r which are named the *output pools*. When a call to `/dev/urandom` is made, data is generated from the pool S_u and when a call to `/dev/random` is made, data is generated from the pool S_r.

[8] All descriptions were done by source code analysis. We refer to version 3.7.8 of the Linux kernel.

Functions refresh and next. There are two refresh functions, $\mathsf{refresh}_i$ that initializes the internal state and $\mathsf{refresh}_c$ that updates it continuously. There are two next functions, next_u for `/dev/urandom` and next_r for `/dev/random`.

Mixing Function. The PRNG uses a *mixing function* M, described in Section 5.6, to mix new data in the input pool and to transfer data between the pools.

Entropy Estimator. The PRNG uses an *entropy estimator*, described in Section 5.4, to estimate the entropy of the collected input and to continuously estimates the entropy of the pools. With these estimations, the PRNG controls the transfers between the pools and how new input is collected. This is illustrated in Figure 3 and described in details in Section 5.3 but at high level, the main principles are:

- New inputs are ignored when the input pool contains enough entropy. Otherwise, the estimated entropy of the input pool is increased with new input.
- Entropy estimation of the output pool is decreased on generation.
- Data is transfered from the input pool to the output pools if they require entropy.
- When the pools do not contain enough entropy, no output can be generated with `/dev/random` and it blocks whereas `/dev/urandom` always generates output.

The technical internal parameters that are the entropy estimations are named E_i (entropy estimator of S_i), E_u (of S_u), E_r (of S_r).

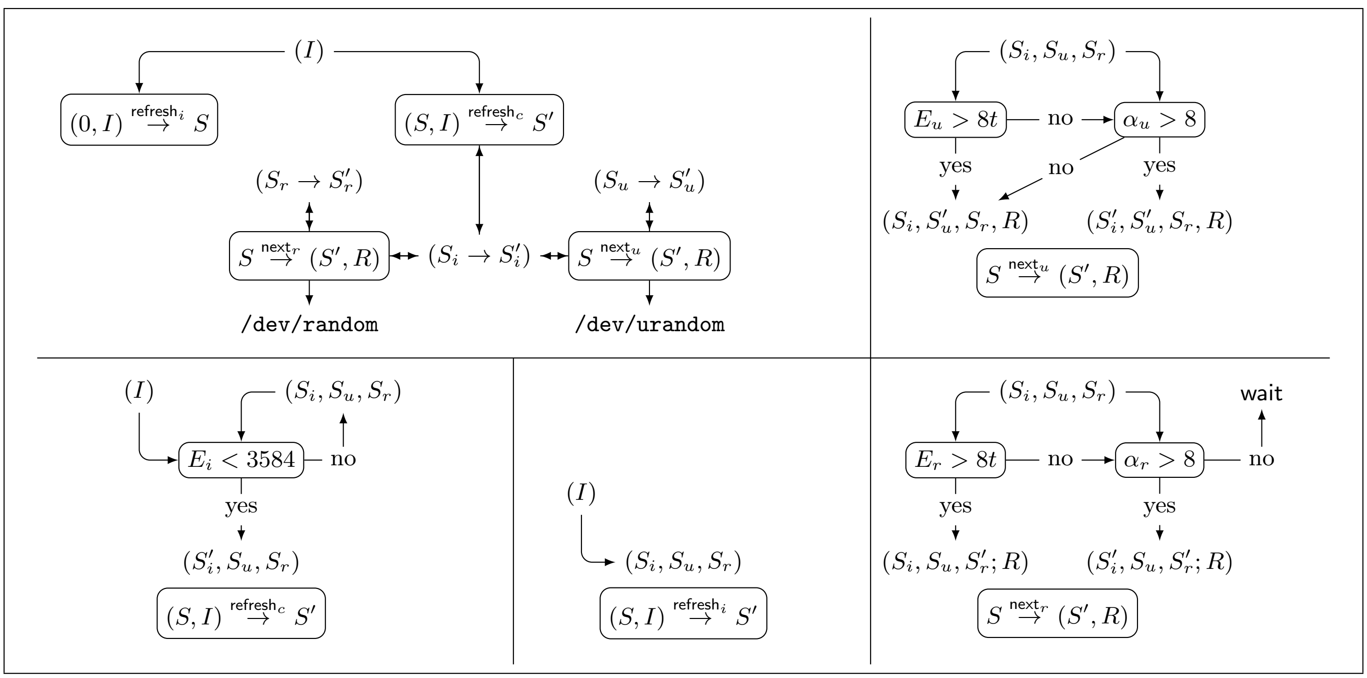

Fig. 3. Relations between functions and pools for LINUX

5.2 The $\mathsf{refresh}_i$ and $\mathsf{refresh}_c$ Functions

The PRNG LINUX contains two refresh functions. A first refresh function, $\mathsf{refresh}_i$, is used to generate the first internal state of the PRNG and the second one, $\mathsf{refresh}_c$, is used to refresh continuously the PRNG with new input.

Internal State Initialisation with $\mathsf{refresh}_i$. To generate the first internal state with $\mathsf{refresh}_i$, LINUX collects device-specific data using a built-in function called add_device_randomness and refreshes S_i and S_n with them. The data is derived from system calls, a call to variable jiffies, which gives the number of CPU cycles since system start-up and is represented by 32 bits, and a call to system function get_cycles, that gives the number of clock ticks since system start-up, which also returns 32 bits. The two values are xor-ed together, giving a new 32-bit input data that is generated twice for S_i and S_n and mixed for each pool. Then LINUX collects system data and refreshes the three pools S_i, S_n and S_b with them using built-in function init_std_data. The data is derived from system calls, a call to function ktime_get_real, which returns 64 bits and a call to function utsname, which returns 3120 bits. The two are concatenated, giving 3184 bits. This input data is generated for each pool and mixed with M, implemented in the built-in function mix_pool_bytes. Finally, the generated input is $I = (\mathsf{utsname}\|\mathsf{ktime\_get\_real}\|\mathsf{get\_cycles} \oplus \mathsf{jiffies})$ for S_i and S_n, and $I = (\mathsf{utsname}\|\mathsf{ktime\_get\_real})$ for S_r. In all cases, $\mathsf{refresh}_i(0, I) = \mathsf{M}(0, I)$. The entropy estimator is not used during this process, so $E_i = E_u = E_r = 0$.

Algorithm 1 Internal State Initialisation with $\mathsf{refresh}_i$

Require: $I_1 = [\mathsf{utsname}\|\mathsf{ktime\_get\_real}\|\mathsf{get\_cycles} \oplus \mathsf{jiffies}], I_2 = [\mathsf{utsname}\|\mathsf{ktime\_get\_real}]$, $S = \varnothing$
Ensure: $S = (S_i, S_u, S_r)$
  $S_i = \mathsf{M}(I_1, 0)$
  $S_r = \mathsf{M}(I_2, 0)$
  $S_u = 0$
  return $S = (S_i, S_u, S_r)$

Internal State Update with $\mathsf{refresh}_c$. The $\mathsf{refresh}_c$ function uses system events that are collected by three built-in functions: add_input_randomness, add _interrupt_randomness and add_disk_randomness. All of them call another built-in function, add_timer _randomness, which builds a 96 bits input data containing the collected event mapped to a specific value num coded in 32 bits, concatenated with jiffies and get_cycles. Finally, the generated input is then given by $I = [\mathsf{num}\|\mathsf{jiffies}\|\mathsf{get\_cycles}]$. If the estimated entropy is above the default value 3584, this input is ignored (except 1 input over 4096). The entropy estimator Ent described in Section 5.4 is used to estimate the entropy of the new input and is added to E_i.

Algorithm 2 Internal State Update with $\mathsf{refresh}_c$

Require: $I = [\mathsf{num}\|\mathsf{jiffies}\|\mathsf{get\_cycles}]$, $S = (S_i, S_u, S_r)$
Ensure: $S' = (S'_i, S'_u, S'_r)$
  if $E_i \geq 3584$ **then**
    $S'_i = S_i$
  else
    $e = \mathsf{Ent}(I)$
    $S'_i = \mathsf{M}(I, S_i)$
    $E_i = e + E_i$
  end if
  $(S'_u, S'_r) = (S_u, S_r)$
  return $S' = (S'_i, S'_u, S'_r)$

Remark 2. Starting from version 3.6.0 of the kernel, LINUX involves a particular behavior of add_interrupt _randomness which collects system events and gather them in a dedicated 128 bits pool fast_pool without calling add_timer _randomness. In this case, the input is $I = \mathsf{fast\_pool}$.

For all these inputs, $\mathsf{refresh}_c(S_i, I) = \mathsf{M}(S_i, I)$ and LINUX estimates the entropy of the data collected by add_timer _randomness and estimates every input collected from fast_pool to 1 bit.

Remark 3. Starting from version 3.2.0 of the kernel, for both /dev/urandom and /dev/random, there is an additional input specific for x86 architectures for which a hardware random number generator is available. In this case, the output of the PRNG is mixed with M when this generator is used for $\mathsf{refresh}_i$ and the output is mixed with the output of LINUX when used with next. For this specific architecture, denoting I_{hd} the input generated by the hardware random number generator, $\mathsf{refresh}_i(S_i, I_{hd}) = \mathsf{M}(S_i, I_{hd})$ and $\mathsf{next}_{hd}(S) = [I_{hd}||\mathsf{next}(S)]$.

5.3 The next_u and next_r Functions

The next functions use built-in functions random_read and urandom_read that are user interfaces to read data from /dev/random and /dev/urandom, respectively. A third kernel interface, get_random_bytes(), allows to read from /dev/urandom. The three rely on the same built-in function extract_buf that calls the mixing function M, a hash function H (the SHA1 function) and a folding function $\mathsf{F}(w_0, \cdots, w_4) = (w_0 \oplus w_3, w_1 \oplus w_4, w_{2_{[0\cdots15]}} \oplus w_{2_{[16\cdots31]}})$.

PRNG Ouput with /dev/random. Let us describe the transfers when t bytes are requested from the blocking pool. If $E_r \geq 8t$, then the output is generated directly from S_r: LINUX first calculates a hash across S_r, then mixes this hash back with S_r, hashes again the output of the mixing function and folds the result in half, giving $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_r, \mathsf{H}(S_r))$ and $S'_r = \mathsf{M}(S_r, \mathsf{H}(S_r))$. This decreases E_r by $8t$ and the new value is $E_r - 8t$. If $E_r < 8t$, then depending on E_i, data is transferred from S_i to S_r. Let $\alpha_r = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor)$.

- If $\alpha_r \geq 8$, then α_r bytes are transferred between S_i and S_r (so at least 8 bytes and at most 128 bytes are transferred between S_i and S_r, and S_i can contain 0 entropy. The transfer is made in two steps: first LINUX generates from S_i an intermediate data $T_i = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i, \mathsf{H}(S_i))$ and then it mixes it with S_r, giving the intermediate states $S'_i = \mathsf{M}(S_i, \mathsf{H}(S_i))$ and $S^*_r = \mathsf{M}(S_r, T_i)$. This decreases E_i by $8\alpha_r$ and increases E_r by $8\alpha_r$. Finally LINUX outputs t bytes from S^*_r, this produces the final output pool $S'_r = \mathsf{M}(S^*_r, \mathsf{H}(S^*_r))$ and $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S^*_r, \mathsf{H}(S^*_r))$. This decreases E_r by $8t$.
- If $\alpha_r < 8$, then LINUX blocks and waits until S_i gets refreshed with I and until $\alpha_r \geq 8$.

Algorithm 3 Output Generation with next_r

Require: t, $S = (S_i, S_u, S_r)$
Ensure: R, $S' = (S'_i, S'_u, S'_r)$
  $\alpha_r = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor)$
  if $\alpha_r \geq 8$ **then**
    $T_i = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i, \mathsf{H}(S_i))$
    $S'_i = \mathsf{M}(S_i, \mathsf{H}(S_i))$
    $S^*_r = \mathsf{M}(S_r, T_i)$
    $E_i = E_i - 8\alpha_r$
    $E_r = E_r + 8\alpha_r$
    $S'_r = \mathsf{M}(S^*_r, \mathsf{H}(S^*_r))$
    $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S^*_r, \mathsf{H}(S^*_r))$
    $E_r = E_r - 8t$
  else
    Blocks until $\alpha_r \geq 8$
  end if
  $S'_u = S_u$
  return R, $S' = (S'_i, S'_u, S'_r)$

PRNG Ouput with /dev/urandom. Similarly, let us describe the transfers when t bytes are requested from the non-blocking pool. If $E_u \geq 8t$ then LINUX applies the same process as in the non-blocking case, outputs $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_u, \mathsf{H}(S_u)))$ and sets $S'_u = \mathsf{M}(S_u, \mathsf{H}(S_u))$. If $E_u < 8t$ then LINUX behaves differently. Let $\alpha_u = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor - 16)$:

– If $\alpha_u \geq 8$, the process is the same as in the non-blocking case, but with S_u, E_u and α_u instead of S_r, E_r and α_r.
– If $\alpha_u < 8$, then LINUX outputs the requested bytes from S_u without transferring data from S_i. Hence LINUX behaves as if $E_u \geq 8t$: $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_u, \mathsf{H}(S_u))$, and $S'_u = \mathsf{M}(S_u, \mathsf{H}(S_u))$. This decreases E_u by $8t$ and the new value is 0.

Algorithm 4 Output Generation with next_u

Require: t, $S = (S_i, S_u, S_r)$
Ensure: R, $S' = (S'_i, S'_u, S'_r)$
  $\alpha_u = \min(\min(\max(t, 8), 128), \lfloor E_i/8 \rfloor - 16)$
  if $\alpha_u \geq 8$ **then**
    $T_i = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i, \mathsf{H}(S_i))$
    $S'_i = \mathsf{M}(S_i, \mathsf{H}(S_i))$
    $S^*_u = \mathsf{M}(S_u, T_i)$
    $E_i = E_i - 8\alpha_u$
    $E_u = E_u + 8\alpha_u$
    $S'_u = \mathsf{M}(S^*_u, \mathsf{H}(S^*_u))$
    $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S^*_u, \mathsf{H}(S^*_u))$
    $E_u = E_u - 8t$
  else
    $R = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_u, \mathsf{H}(S_u))$
    $E_u = 0$
  end if
  $S'_r = S_r$
  return R, $S' = (S'_i, S'_u, S'_r)$

This illustrates the difference between `/dev/urandom` and `/dev/random`: If the estimated entropy of the blocking pool S_r is less than $8t$ and no transfer is done, then `/dev/random` blocks, whereas `/dev/urandom` does never block and outputs the requested t bytes from the non-blocking pool S_u.

5.4 The Entropy Estimator

A built-in estimator Ent is used to give an estimation of the entropy of the input data used to refresh S_i. It is implemented in function add_timer_randomness which is used to refresh the input pool. A timing t_n is associated with each event (system or user call) that is used to refresh the internal state. Entropy is estimated when new input data is used to refresh the internal state, entropy is not estimated using input distribution but only using the timings of the data. A description of the estimator is given in [GPR06], [LRSV12] and [GLSV12]. The estimator takes as input a sequence of inputs $I_i = [\mathsf{num} \| \mathsf{jiffies} \| \mathsf{get\_cycles}]$, it calculates differences between timings of events, where t_0, t_1, t_2, ... are the jiffies associated with each input: $\delta_i = t_i - t_{i-1}$, $\delta_i^2 = \delta_i - \delta_{i-1}$, $\delta_i^3 = \delta_i^2 - \delta_{i-1}^2$. Then, it calculates $\Delta_i = \min(|\delta_i|, |\delta_i^2|, |\delta_i^3|)$ and finally applies a logarithmic function to give the estimated entropy $H_i = 0$ if $\Delta_i < 2$, $H_i = 11$ if $\Delta_i > 2^{12}$, and $H_i = \lfloor \log_2(\Delta_i) \rfloor$ otherwise.

5.5 The Folding and the Hash Functions

The folding function F and the hash function H are used when random bytes are generated by LINUX and when data is transferred from S_i to S_r or S_u. The folding function is implemented in built-in function extract_buf. It take as input five 32-bit words and output 80 bits of data. This function F is defined by $\mathsf{F}(w_0, w_1, w_2, w_3, w_4) = (w_0 \oplus w_3, w_1 \oplus w_4, w_{2_{[0\cdots15]}} \oplus w_{2_{[16\cdots31]}})$, where w_i for $i \in \{0, \ldots, 4\}$ are the input

Algorithm 5 Entropy Estimator Ent

Require: $I_i = [\mathsf{num}\|\mathsf{jiffies}\|\mathsf{get\_cycles}]$
Ensure: $H_i = \mathsf{Ent}(I_i)$
  $t_i = \mathsf{jiffies}$
  $\delta_i = t_i - t_{i-1}$
  $\delta_i^2 = \delta_i - \delta_{i-1}$
  $\delta_i^3 = \delta_i^2 - \delta_{i-1}^2$
  $\Delta_i = \min(|\delta_i|, |\delta_i^2|, |\delta_i^3|)$
  if $\Delta_i < 2$ **then** $H_i = 0$
  if $\Delta_i > 2^{12}$ **then** $H_i = 11$
  else $H_i = \lfloor \log_2(\Delta_i) \rfloor$
  return $H_i = \mathsf{Ent}(I_i)$

words.

The hash function H is implemented in the built-in function extract_buf by a call to a Linux system function sha_transform that implements function SHA1, defined in [SHA95].

5.6 The Mixing Function

The Mixing function M is the core of LINUX PRNG. It is implemented in the built-in function mix_pool_bytes. It is used in two contexts, once to refresh the internal state with new intput and secondly to transfer data between the input pool and the output pools. We give a complete description of M as it is used to refresh the input pool S_i, its description when it is used to transfer data between pools differs only from internal parameters.

The function M takes as input I of size one byte, the input pool S_i that is considered as a table of 128 32-bits words. It selects 7 words in S_i and mixes them with I and replace one word of S_i with the result. The pool S_i therefore maintains an internal parameter, named k, which is used to select the word that will be modified. Another internal parameter, named d, is used in function M. This parameter is a multiple of 7 used in a rotation done at word level. We name the rotation of d bits R_d. The mixing function involves the following operations:

- The byte containing the entropy source is converted into a 32-bit word, using standard C implicit cast, and rotated by d bits. Before initialization, $d = 0$, and each time the mixing function M is used, d is incremented using k : if $k = 0 \bmod 128$ then $d = d + 14 \bmod 32$ and $d = d + 7 \bmod 32$ otherwise.
- The obtained word is xor-ed with words from the pool. If we note $S_0, \ldots, S_{127}$ the words of S_i, chosen words will be $S_{k+j \bmod 128}$ for $j \in \{0, 1, 25, 51, 76, 103\}$[9].
- The obtained word is mixed with a built-in table (named *twist table*). This table contains the binary representations of the monomials $\{0, \alpha^{32*j}\}, j = 1, \ldots 7$, in the field $(\mathbb{F}_2)/(Q)$, where $Q(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ is the CRC32 polynomial used for Ethernet protocol [Koo02]. Denoting the primitive element α, this operation can be described as $W \to W.\alpha^3 + \mathsf{R}(\mathsf{Q}(W, \alpha^{29}).\alpha^{32}, Q)$, where $\mathsf{Q}(A, B)$ (resp. $\mathsf{R}(A, B)$) the quotient (resp. the remainder) in polynomial division A/B.
- Then the word at index k in S_i is replaced by the previously generated word and k is incremented.

5.7 Distributions Used for Attacks

Distributions Used in Attacks based on the Entropy Estimator As shown in Section 5.4, LINUX uses an internal *Entropy Estimator* on each input that continuously refreshes the internal state of the PRNG. We show that this estimator can be fooled in two ways. First, it is possible to define a distribution of zero entropy

[9] Similarly, the words chosen from S_r and S_u will be $S_{k+j \bmod 32}$ for $j \in \{0, 1, 7, 14, 20, 26\}$.

Algorithm 6 The Mixing function

Require: I, $S = (S_0, \ldots, S_k, \ldots, S_{127})$
Ensure: S'
$W = \mathsf{R}_d[0||I]$
if $k = 0 \bmod 128$ **then** $d = d + 14 \bmod 32$ **else** $d = d + 7 \bmod 32$ **end if**
$W = W \oplus S_{k+j \bmod 128}$, $j \in \{0, 1, 25, 51, 76, 103\}$
$W = W.\alpha^3 + \mathsf{R}(\mathsf{Q}(W, \alpha^{29}).\alpha^{32}, Q)$
$S'_k = W$
$k = k + 1$
return $S = (S_0, \ldots, S'_k, \ldots, S_{127})$

that the estimator will estimate of high entropy, secondly, it is possible to define a distribution of arbitrary high entropy that the estimator will estimate of zero entropy. This is due to the estimator conception: as it considers the timings of the events to estimate their entropy, regular events (but with unpredictable data) will be estimated with zero entropy, whereas irregular events (but with predictable data) will be estimated with high entropy. These two distributions are given in Lemma 3 and 4.

Lemma 3. *There exists a stateful distribution $\mathcal{D}_0$ such that $\mathbf{H}_\infty(\mathcal{D}_0) = 0$, whose estimated entropy by* LINUX *is high.*

Proof. Let us define the 32-bits word distribution $\mathcal{D}_0$. On input a state i, $\mathcal{D}_0$ updates its state to $i+1$ and outputs a triple $(i+1, [W_1^i, W_2^i, W_3^i]) \xleftarrow{\$} \mathcal{D}_0(i)$, where $W_1^0 = 2^{12}, W_1^i = \lfloor \cos(i).2^{20} \rfloor + W_1^{i-1}$, $W_2^i = W_3^i = 0$. For each state, $\mathcal{D}_0$ outputs a 12-bytes input containing 0 bit of random data, we have $\mathbf{H}_\infty(\mathcal{D}_0) = 0$ conditioned on the previous and the future outputs (*i.e.* $\mathcal{D}_0$ is legitimate only with $\gamma_i = 0$ for all i). Then $\Delta_i > 2^{12}$ and $H_i = 11$. □

Lemma 4. *There exists a stateful distribution $\mathcal{D}_1$ such that $\mathbf{H}_\infty(\mathcal{D}_1) = 64$, whose estimated entropy by* LINUX *is null.*

Proof. Let us define the 32-bits word distribution $\mathcal{D}_1$. On input a state i, $\mathcal{D}_1$ updates its state to $i+1$ and outputs a triple: $(i+1, [W_1^i, W_2^i, W_3^i]) \xleftarrow{\$} \mathcal{D}_1(i)$, where $W_i = i, W_2 \xleftarrow{\$} \mathcal{U}_{32}$ and $W_3 \xleftarrow{\$} \mathcal{U}_{32}$. For each state, $\mathcal{D}_1$ outputs a 12-bytes input containing 8 bytes of random data, we have $\mathbf{H}_\infty(\mathcal{D}_1) = 64$ conditioned on the previous and the future outputs (*i.e.* $\mathcal{D}_1$ is legitimate with $\gamma_i = 64$ for all i). Then $\delta_i = 1$, $\delta_i^2 = 0$, $\delta_{i-1}^2 = 0$, $\delta_i^3 = 0$, $\Delta_i = 0$ and $H_i = 0$. □

Distribution Used in Attack based on the Mixing Function As shown in Section 5.6, LINUX uses an internal Mixing function M, used to refresh the internal state with new intput and to transfer data between the pools. It is possible to define a distribution of arbitrary high entropy for which the Mixing function is completely counter productive, i.e. the entropy of the internal state does not increase, whatever the size of the input is. This is due to the conception of the Mixing function and its linear structure. This distribution is given in Lemma 5.

Lemma 5. *There exists a stateful distribution $\mathcal{D}_2$ such that $\mathbf{H}_\infty(\mathcal{D}_2) = 1$, for which $\mathbf{H}_\infty(S) = 1$ after t* refresh*, for arbitrary high t.*

Proof. Let us define the byte distributions $\mathcal{B}_{i,b}$ and $\mathcal{B}_{i,\$}$:

$$\mathcal{B}_{i,b} = \{(0, \cdots, b, \cdots, 0), b_i \leftarrow b, b_j = 0 \text{ if } i \neq j\}$$
$$\mathcal{B}_{i,\$} = \{(b_0, \cdots, b_7), b_i \xleftarrow{\$} \{0,1\}, b_j = 0 \text{ if } i \neq j\}$$

Let us define the 12 bytes distribution $\mathcal{D}_2$. On input a state i, $\mathcal{D}_2$ updates its state to $i+1$ and outputs 12 bytes:

$$\begin{array}{l}(i+1,[B_0^i,\ldots,B_{11}^i]) \xleftarrow{\$} \mathcal{D}_2(i), \text{where } B_4^{10i} \leftarrow \mathcal{B}_{7,\$},\\ B_5^{10i} \leftarrow \mathcal{B}_{3,b}, B_4^{10i+2} \leftarrow \mathcal{B}_{2,b}, B_7^{10i+4} \leftarrow \mathcal{B}_{5,b},\\ B_6^{10i+6} \leftarrow \mathcal{B}_{1,b}, B_{10}^{10i+8} \leftarrow \mathcal{B}_{0,b}, \text{with } b = B_{4,7}^i\end{array}$$

For each state i, $\mathcal{D}_2$ outputs a 12-bytes input containing 1 bit of random data (for $i = 0 \bmod 10$) or 0 bit of random data (for $i \neq 0 \bmod 10$).
If $d = 0$, $k = 127$ and S is known, and noting $S^t = \mathsf{refresh}(S, \mathsf{refresh}(S^{t-1},[B_0^{t-1},\ldots,B_{11}^{t-1}]))$, $S^t = S_0^t,\ldots,S_{127}^t$, then S^t contains 1 random bit in word S_{127}^t, at position 10, for all t. Distribution outputs are illustrated in Figure 4. □

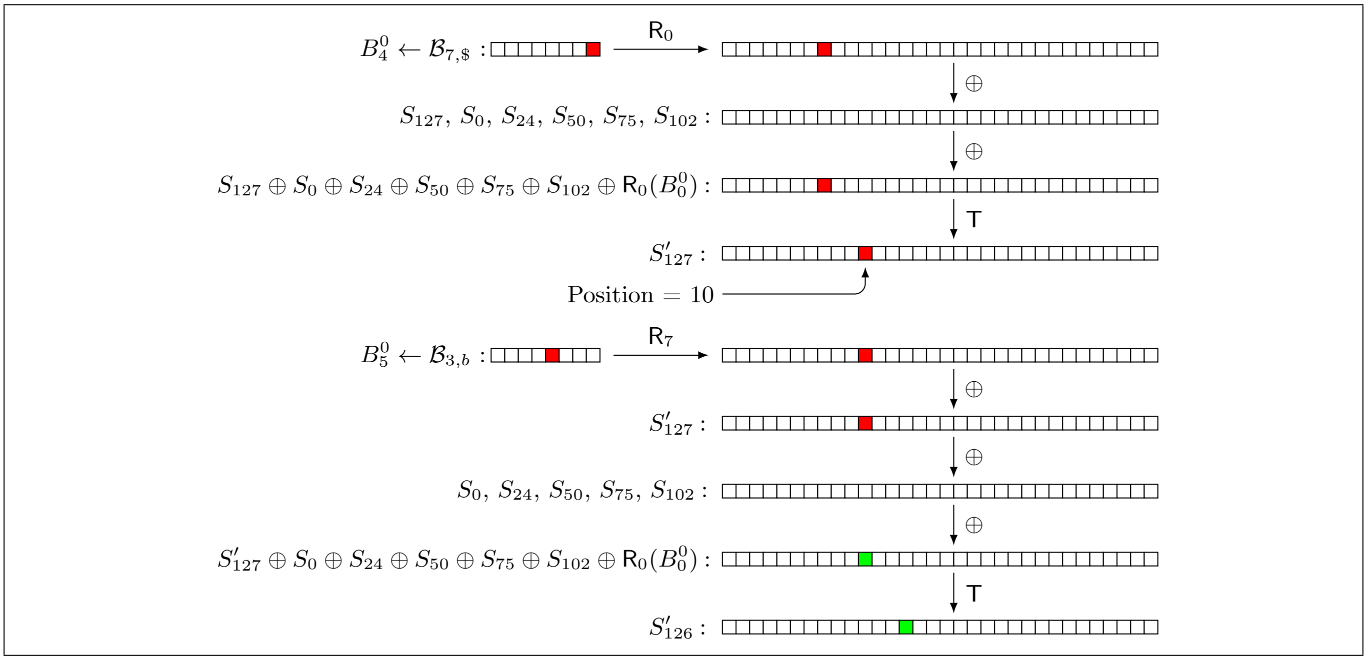

Fig. 4. Distribution $\mathcal{D}_2$: output of B_4^0 and B_5^0

5.8 Attack Descriptions

In this section we describe attacks on LINUX that prove Theorem 3. The first two attacks use distributions that fools the PRNG Entropy Estimator and the last attack uses the distribution for which the Mixing function is counter productive. For this last attack, we show indeed that LINUX is not even backward secure.

Attacks Based on the Entropy Estimator As shown in Section 5.7, it is possible to build a distribution $\mathcal{D}_0$ of null entropy for which the estimated entropy is high (*cf.* Lemma 3) and a distribution $\mathcal{D}_1$ of high entropy for which the estimated entropy is null (*cf.* Lemma 4). It is then possible to mount attacks on both `/dev/random` and `/dev/urandom`, which show that these two generators are not robust.

`/dev/random` **is not robust.** Let us consider an adversary $\mathcal{A}$ against the robustness of the generator `/dev/random`, and thus in the game $\mathsf{ROB}(\gamma^*)$, that makes the following oracle queries: one get-state, several next-ror, several $\mathcal{D}$-refresh and one final next-ror.

Then the state (S_i, S_r, S_u), the parameters k, d, E_i, E_u, E_r and the counter c defined in $\mathsf{ROB}(\gamma^*)$ evolve the following way:

- get-state: After a state compromise, $\mathcal{A}$ knows all parameters (but needs S_i, S_r, E_i, E_r) and $c = 0$.
- next-ror: After $\lfloor E_i/10 \rfloor + \lfloor E_r/10 \rfloor$ queries to next-ror, $E_i = E_r = 0$, $\mathcal{A}$ knows S_i and S_r and $c = 0$.
- $\mathcal{D}$-refresh: In a first stage, $\mathcal{A}$ refreshes LINUX with input from $\mathcal{D}_0$. After 300 queries, $E_i = 3584$ and $E_r = 0$. $\mathcal{A}$ knows S_i and S_r and $c = 0$.

  In a second stage, $\mathcal{A}$ refreshes LINUX with input $J \stackrel{\$}{\leftarrow} \mathcal{U}_{128}$. As $E_i = 3584$, these inputs are ignored as long as I contains less than 4096 bytes. After 30 queries, $\mathcal{A}$ knows S_i and S_r and $c = 3840$.
- next-ror: Since $E_r = 0$, a transfer is necessary between S_i and S_r before generating R. Since $E_i = 3584$, then $\alpha_r = 10$, such a transfer happens. But as $\mathcal{A}$ knows S_i and S_r, then $\mathcal{A}$ knows R.

Therefore, in the game $\mathsf{ROB}(\gamma^*)$ with $b = 0$, $\mathcal{A}$ obtains a 10-bytes string in the last next-ror-oracle that is predictable, whereas when $b = 1$, this event occurs only with probability 2^{-80}. It is therefore straightforward for $\mathcal{A}$ to distinguish the real and the ideal world.

`/dev/urandom` is not robust. Similarly, let us consider an adversary $\mathcal{A}$ against the robustness of the generator `/dev/urandom` in the game $\mathsf{ROB}(\gamma^*)$ that makes the following oracle queries: one get-state that allows it to know S_i, S_u, E_i, E_u; $\lfloor E_i/10 \rfloor + \lfloor E_u/10 \rfloor$ next-ror, making $E_i = E_u = 0$; 100 $\mathcal{D}$-refresh with $\mathcal{D}_1$; and one next-ror, so that R will only rely on S_u as no transfer is done between S_i and S_u since $E_i = 0$. Then $\mathcal{A}$ is able to generate a predictable output R and to distinguish the real and the ideal worlds in $\mathsf{ROB}(\gamma^*)$.

Attack based on the Mixing Function. In [LRSV12], a proof of state entropy preservation is given for one iteration of the mixing function M, assuming that the input and the internal state are independent, that is: $\mathbf{H}_\infty(\mathsf{M}(S, I)) \geq \mathbf{H}_\infty(S)$ and $\mathbf{H}_\infty(\mathsf{M}(S, I)) \geq \mathbf{H}_\infty(I)$. We show that without that independence assumption and with more than one iteration of M, the LINUX PRNG does not recover from state compromise. This contradicts the backward security and therefore the robustness property.

LINUX is not backward secure. As shown in Section 5.7, with Lemma 5, it is possible to build an input distribution $\mathcal{D}_2$ with arbitrary high entropy such that, after several $\mathcal{D}$-refresh, $\mathbf{H}_\infty(S) = 1$. Let us consider an adversary $\mathcal{A}$ that generates an input data of distribution $\mathcal{D}_2$, and that makes the following oracle queries: set-refresh, and γ^* calls to $\mathcal{D}$-refresh followed by many calls to next-ror. Then the state (S_i, S_r, S_u), the parameters k, d, E_i, E_u, E_r and the counter c of $\mathsf{BWD}(\gamma^*)$ evolve the following way:

- set-refresh: $\mathcal{A}$ sets $S_i = 0$, $S_r = S_u = 0$, $d = 0$ and $k = 127$, and $c = 0$.
- $\mathcal{D}$-refresh: $\mathcal{A}$ refreshes LINUX with $\mathcal{D}_2$. After γ^* oracle queries, until $c \geq \gamma^*$, the new state still satisfies $\mathbf{H}_\infty(S) = 1$.
- next-ror: Since $\mathbf{H}_\infty(S) = 1$, $\mathbf{H}_\infty(R) = 1$.

Therefore, in the game $\mathsf{BWD}(\gamma^*)$ with $b = 0$, $\mathcal{A}$ always obtains an output in the last next-ror query with $\mathbf{H}_\infty(R) = 1$, whereas in $b = 1$, this event occurs only with negligible probability. It is therefore straightforward for $\mathcal{A}$ to distinguish the real and the ideal world.

6 Benchmarks Between LINUX and our Construction

In this section we present efficiency benchmarks between our construction $\mathcal{G}$ and LINUX. These benchmarks are based on a very optimistic hypothesis concerning LINUX and even with this hypothesis, our construction $\mathcal{G}$ appears to be more efficient. As shown in Section 6.4, a complete internal state accumulation is on average two times faster for $\mathcal{G}$ than for LINUX and as shown in Section 6.5, a 2048-bits key generation is on average ten times faster for $\mathcal{G}$ than for LINUX.

6.1 Security Bounds

We will now instantiate our main construction presented in Corollary 1 for various values of "security parameter" k using AES_128 in counter mode to out pseudorandom generator $\mathbf{G}$. Namely, we set $m = \ell = 128$ (recall, m is the PRG input size, and ℓ in the output size), and let $\mathbf{G}(U) = \mathrm{AES}_U(0) \ldots \mathrm{AES}_U(i-1)$, where $i = \lceil (n+128)/128 \rceil$ is the number of calls to AES_128 to get one 128-bit output. Recall also from Corollary 1 that we set the state length $n = m + 9k + 1 = 9k + 129$, which gives $i = 2 + \lceil (9k+1)/128 \rceil$.

We are now ready to instantiate Corollary 1 (for various values of k), except we need to set the security ε_{prg} of our counter-mode PRG in terms of the security of AES. This turns out to be a slightly subtle issue, which we discuss at the end of this section, in part because it is based on assumptions, and also because the "provable term" $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$ seems to be overly pessimistic and does not correspond to an actual attack. Hence, for now we will optimistically assume that, for the values of security parameter k we consider, we have $\varepsilon_{comp} \leq \varepsilon_{stat} = 2^{-k}$, so that $\varepsilon = \varepsilon_{comp} + \varepsilon_{stat} \approx \varepsilon_{stat} = 2^{-k}$.

With this in mind, we will consider setting the security level k to three values: 40 ("medium"), 50 ("high") and 64 ("unbreakable"). Recalling that $n = m + 9k + 1 = 9k + 129$, $\gamma^* = m + 8k + 1 = 8k + 129$, and $i = 2 + \lceil (9k+1)/128 \rceil$, we get:

- *Medium Security* $\varepsilon_{stat} = 2^{-40}$ ($q_R = q_S = q_{\mathcal{D}} = 2^{40}$): we get $n = 489$, $\gamma^* = 449$, $i = 5$.
- *High Security* $\varepsilon_{stat} = 2^{-50}$ ($q_R = q_S = q_{\mathcal{D}} = 2^{50}$): we get $n = 579$, $\gamma^* = 529$, $i = 6$.
- *Unbreakable Security* $\varepsilon_{stat} = 2^{-64}$ ($q_R = q_S = q_{\mathcal{D}} = 2^{64}$): we get $n = 705$, $\gamma^* = 641$, $i = 7$.

To perform benchmarks between LINUX and generator $\mathcal{G}$, we instantiated $\mathcal{G}$ with AES function in counter mode and the fields $\mathbb{F}_{2^{489}}$ (defined by the polynom $X^{489} + X^{83} + 1$), $\mathbb{F}_{2^{579}}$ (defined by the polynom $X^{579} + X^{12} + X^9 + X^7 + 1$) and $\mathbb{F}_{2^{705}}$ (defined by the polynom $X^{705} + X^{17} + 1$). We set the output size of AES function equal to 128 bits and we describe this instantiation with $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$, where:

- $\mathsf{setup} = (X, X') \xleftarrow{\$} \{0,1\}^{489+489}$ (*resp.* $\{0,1\}^{579\times579}$, $\{0,1\}^{705\times705}$);
- $\mathsf{refresh}(S, I) = S \cdot X + I \in \mathbb{F}_{2^{489}}$ (*resp.* $\mathbb{F}_{2^{579}}$, $\mathbb{F}_{2^{705}}$);
- $\mathsf{next}(S) : U = [S \cdot X']_1^{128}$, $(S', R) = (\mathrm{AES}_U(0), \ldots, \mathrm{AES}_U(4))$ (*resp.* $\mathrm{AES}_U(5)$), $\mathrm{AES}_U(6)$).

Computational Term ε_{comp}. We now come back to estimating the computational term $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$, and our optimistic assumption that $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg} \leq \varepsilon_{stat} = 2^{-k}$, which is equivalent to $\varepsilon_{prg} \leq 2^{-2k-1}$. Since we also want the running time $t \geq q_R = 2^k$, we essentially need our pseudorandom generator $\mathbf{G}$ to be $(2^k, 2^{-2k})$-secure. However, it is easy to notice that any $(2^k, \varepsilon_{prg})$-secure PRG with an m-bit key cannot have security $\varepsilon_{prg} < 2^{k-m}$, since the attacker in time 2^k can exhaustively try 2^k out of 2^m key to achieve advantage 2^{k-m}. This means that we need to have $2^{-2k} \geq 2^{k-m}$, or $k \leq m/3$. For example, when using AES_128 in counter mode, this seems to suggest we can have $\varepsilon_{comp} \leq \varepsilon_{stat}$ only for $k \leq 42 = \lfloor 128/3 \rfloor$, which is not the case for our high and unbreakable security settings.

However, we believe that the above analysis is overly pessimistic. Indeed, in theory, if we want to use a given PRG in a stream cipher mode ($(S, R) \leftarrow \mathbf{G}(S)$) for 2^k times, we can only claim "union bound" security $2^k\varepsilon_{prg}$, which, as we saw, is only possible when $k \leq m/3$. Although tight in theory, the bound does not seem to correspond to any *concrete attack* when used with most "real-world" PRGs (such as AES_128 in counter mode). For example, for $k = 64$ (our "unbreakable" setting), the bound $2^k\varepsilon_{prg} \geq 2^{64} \cdot 2^{-64} = 1$, which suggests (if the bound was tight!) that one can break a stream cipher built from AES_128 in the counter mode in 2^{64} queries with advantage 1. However, we are presently not aware of any attack achieving advantage even 2^{-64}, let alone 1. To put it differently, we think that our original assumption that $\varepsilon_{comp} \leq \varepsilon_{stat}$ for $k = 64$ seems reasonable based on our current knowledge, even though theoretical analysis suggests that there is little point to set $k > 42$.

Based on this discussion, we suggest the following recipe when instantiating our construction with a particular pseudorandom $\mathbf{G}$. Instead of directly looking at the term $\varepsilon_{comp} = 2^{k+1}\varepsilon_{prg}$ when examining a candidate value of security parameter k, one should ask the following question instead: *based on the current knowledge, what is the largest value of k (call it k^*) so that no attacker can achieve advantage better than*

2^{-k} *when* $\mathbf{G}$ *is used in the stream cipher mode for* 2^k *times?* When this k^* is determined, there is no point to set $k > k^*$, as this only increases the state length n and degrades the efficiency of the PRNG, without increasing its security ε beyond 2^{-k^*} (as $\varepsilon_{comp} \le 2^{-k^*}$ anyway). However, setting $k \le k^*$ will result in final security $\varepsilon \approx 2^{-k}$ while improving the efficiency of the resulting PRNG (i.e., state length $n = m + 9k + 1$, $\gamma^* = m + 8k + 1$, and the complexity of refresh and next).

With this (somewhat heuristic) recipe, we believe setting $k^* = 64$ was a fair and reasonable choice when using AES_128 in counter mode to implement $\mathbf{G}$.

6.2 Hypothesis

For LINUX, we made the (optimitic) hypothesis that for the given input distribution, the mixing function of LINUX accumulates the entropy in the internal state, that is $\mathbf{H}_\infty(M(S, I)) = \mathbf{H}_\infty(S) + \mathbf{H}_\infty(I)$ if S and I are independent, and that the SHA1 function used for transfer between the pools and output is a perfect extractor, that is $\mathbf{H}_\infty(\mathsf{SHA1}(S_*)) = 160$ if $\mathbf{H}_\infty(S_*) = 160$. Of course, both of these hypotheses are extremely strong, but we make them to achieve the most optimistic (and probably unrealistic!) estimates when comparing LINUX with our construction $\mathcal{G}$.

6.3 Implementation

We implemented LINUX with functions extract_buf and mix_pool_bytes that we extracted from the source code and we implemented $\mathcal{G}$ using fb_mul_lodah and fb_add from RELIC open source library [AG] (that we extended with the fields $\mathbb{F}_{2^{489}}$, $\mathbb{F}_{2^{579}}$ and $\mathbb{F}_{2^{705}}$), aes_setkey_enc and aes_crypt_ctr from PolarSSL open source library [Pol]. CPU cycle count was done using ASM instruction RDTSC. Implementation was done on a x86 Ubuntu workstation. All code was written in C, we used gcc C compiler and linker, code optimization flag O2 was used to build the code.

6.4 Benchmarks on the Accumulation Process

First benchmarks are done on the accumulation process. We simulated a complete accumulation of the internal state for LINUX and $\mathcal{G}$ with an input containing one bit of entropy per byte. For $\mathcal{G}$, by Theorem 2, 8 inputs of size 449 bits (*resp.* 579, 705 bits) are necessary to recover from an internal state compromise, whereas by hypothesis, for LINUX, $\lfloor 160/12 \rfloor = 13$ inputs of size 12 bits are necessary to recover from an internal state compromise and transfers need to be done between the input pool and the output pools.

For LINUX, denoting $S^t = (S_i^t, S_u^t, S_r^t)$, where S_i^t, S_u^t and S_r^t are the successive states of the input pool, the non-blocking output pool and the blocking output pool, respectively, we implemented the following process, starting from a compromised internal state (S_i^0, S_u^0, S_r^0), of size 6144 bits, and using successive inputs of size 12 bytes, that we denote I^t:

1. Refresh S_i^0 with $I^0, \cdots, I^{13}$: $S_i^t = \mathsf{M}(S_i^{t-1}, I^{t-1})$. By hypothesis, $\mathbf{H}_\infty(S_i^{13}) = 168$.
2. Transfer 1024 bits from S_i^{13} to S_r. The transfer is made by blocks of 80 bits, therefore, 13 transfers are necessary. Each transfer is done in two steps: first LINUX generates from S_i^{13} an intermediate data $T_i^{13} = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i^{13}, \mathsf{H}(S_i^{13}))$ and then it mixes it with S_r, giving the new states $S_i^{14} = \mathsf{M}(S_i^{13}, \mathsf{H}(S_i^{13}))$ and $S_r^{14} = \mathsf{M}(S_r^{13}, T_i^{13})$. Then by hypothesis, $\mathbf{H}_\infty(S_r^{13}) = 80$. After repeating these steps 12 times, by hypothesis, $\mathbf{H}_\infty(S_r^{26}) = 1024$.
3. Repeat step 2. for S_u instead of S_r. By hypothesis, $\mathbf{H}_\infty(S_u^{39}) = 1024$.

After this process, by hypothesis, $\mathbf{H}_\infty(S^{39}) = 6144$ is maximal.

For $\mathcal{G}$, denoting S^t the successive states of the internal state, we implemented the following process, starting from a compromised internal state S^0, of size 489 bits (*resp.* 579, 705 bits), and using successive inputs I^t, of size 489 bits (*resp.* 579, 705 bits): Refresh S^0 with $I^0, \cdots, I^7$: $S^i = S^{i-1} \cdot X + I^{i-1}$. After this process, by Theorem 2, $\mathbf{H}_\infty(S^8) = 489$ (*resp.* 579, 705 bits) is maximal.

The number of CPU cycles to perform these processes on LINUX and $\mathcal{G}$ (with internal state size 705 bits) are presented in Figure 5. We first implemented 100 complete accumulations processes for LINUX and $\mathcal{G}$ and we compared one by one each accumulation. As shown on the left part of Figure 5, a complete accumulation in the internal state of $\mathcal{G}$ needs on average two times less CPU cycles than a complete accumulation the internal state of LINUX. Then we analysed one accumulation in detail or LINUX and $\mathcal{G}$. As shown on the right part of Figure 5, a complete accumulation in the internal state of LINUX needs more CPU because of the transfers between the input pool and the two output pools done in steps 2. and 3, it also shows that the refresh function of $\mathcal{G}$ is similar as the Mixing function M of LINUX.

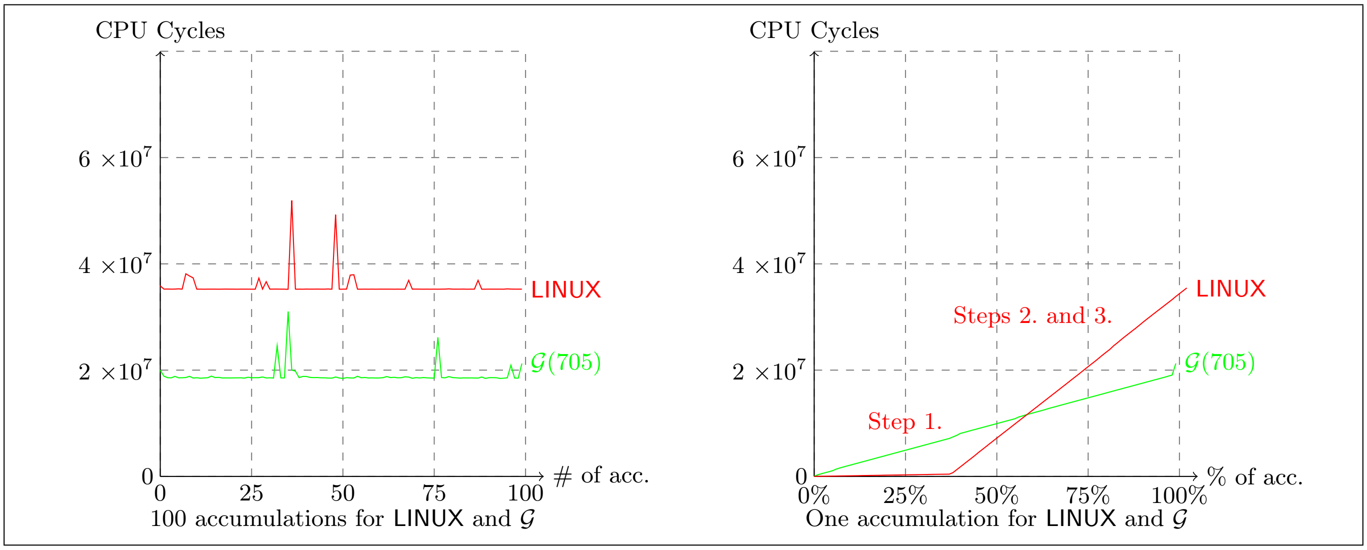

Fig. 5. Accumulation Process

6.5 Benchmarks on the Generation Process

Second benchmarks are done on the generation process. We simulated the generation of 2048-bits keys K for LINUX and $\mathcal{G}$. For $\mathcal{G}$, 16 calls to `next` are necessary, as each call outputs 128 bits. For LINUX, each call to `next` outputs 80 bits, therefore 12 calls are first necessary, then 1024 bits need to be transfered from the input pool to the output pool, then 12 new calls to `next` are necessary.

For LINUX, denoting R^t the successive ouputs, we implemented the following process, starting from an internal state (S_i^0, S_r^0, S_u^0), where we suppose at least 1024 bits of entropy are accumulated in the output pool S_r^0 and 4096 bits of entropy are accumulated in the input pool S_i^0:

1. Set $R^0 = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_r^0, \mathsf{H}(S_r^0))$
2. Repeat step 1. 12 times and set $K^0 = [R^0|| \ldots ||R^{12}]_1^{1024}$.
3. Transfer 1024 bits from S_i^0 to S_r^0. The transfer is made by blocks of 80 bits, therefore, 13 transfers are necessary. Each transfer is done in two steps: first LINUX generates from S_i^0 an intermediate data $T_i^0 = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_i^0, \mathsf{H}(S_i^0))$ and then it mixes it with S_r^0, giving the new states $S_i^1 = \mathsf{M}(S_i^1, \mathsf{H}(S_i^1))$ and $S_r^1 = \mathsf{M}(S_r^0, T_i^0)$. Then by hypothesis, $\mathbf{H}_\infty(S_r^1) = 80$. After repeating these steps 12 times, by hypothesis, $\mathbf{H}_\infty(S_r^{13}) = 1024$.
4. Set $R^{13} = \mathsf{F} \circ \mathsf{H} \circ \mathsf{M}(S_r^{13}, \mathsf{H}(S_r^{13}))$
5. Repeat step 1. 12 times and set $K^1 = [R^{13}|| \ldots ||R^{25}]_1^{1024}$.

6. Set $K = [K^0||K^1]$

After this process, $\mathbf{H}_\infty(K) = 2048$.

For $\mathcal{G}$, we implemented the following process (using the Practical Efficiency Optimization presented in Section 4, starting from an internal state S^0, of size 489 bits (*resp.* 579, 705 bits), where we suppose at least $\gamma^* = 449$ (*resp.* 529, 641 bits) bits of entropy are accumulated:

1. Set $U = [S \cdot X']_1^{128}$ and $(S^1, R^0) = (\mathrm{AES}_U(0), \ldots, \mathrm{AES}_U(4))$ (*resp.* $\mathrm{AES}_U(5)$), $\mathrm{AES}_U(6)$) and set the Boolean flag $\mathsf{last} = \mathsf{true}$.
2. Set $(U, R) = (\mathrm{AES}_U(0), \mathrm{AES}_U(1))$ and set $[S]_1^{128} = U$.
3. Repeat step 2. 14 times.

After this process, $\mathbf{H}_\infty(K) = 2048$.

The number of cycles to perform these processes on LINUX and $\mathcal{G}$ (with internal state size 705 bits) are presented in Figure 6. We first implemented the generation of 100 2048-bits keys and we compared one by one each generation. As shown on the left part of Figure 6, 2048-bits key generation with $\mathcal{G}$ needs on average ten times less CPU cycles than with LINUX. Then we analysed one accumulation in detail or LINUX and $\mathcal{G}$. As shown on the right part of Figure 6, a 2048-bits key generation needs more CPU for LINUX.

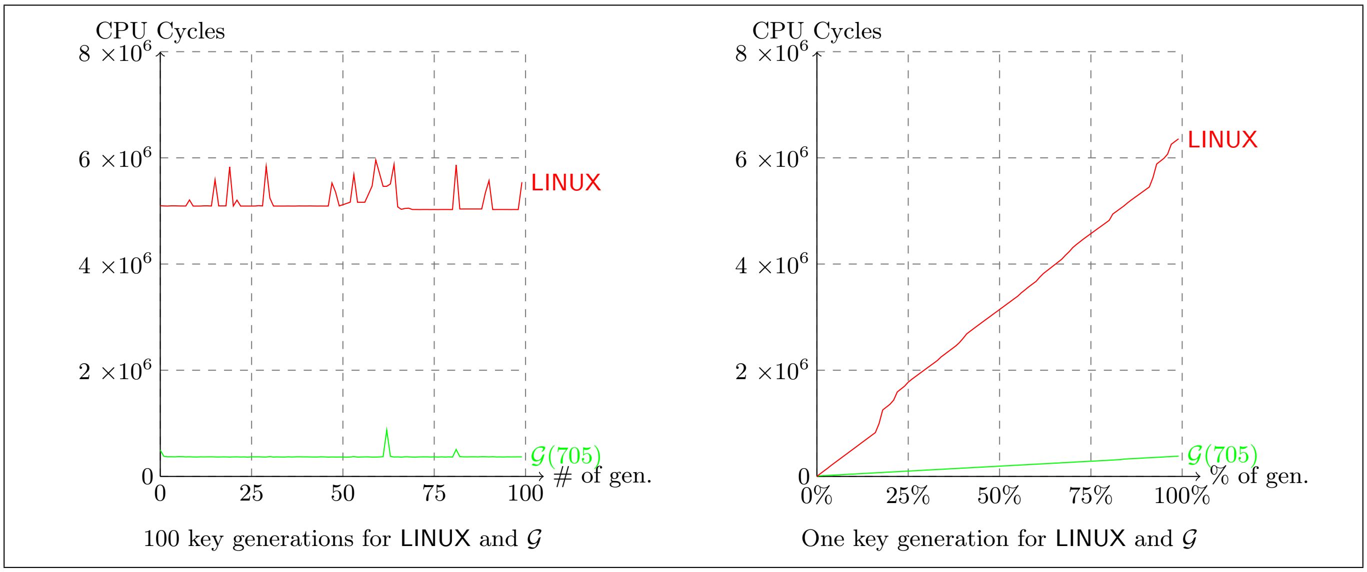

Fig. 6. Generation Process

7 Conclusion

We have proposed a new property for PRNG with input, that captures how it should accumulate the entropy of the input data into the internal state. This property actually expresses the real expected behavior of a PRNG after a state compromise, where it is expected that the PRNG quickly recovers enough entropy. We gave a precise assessment of Linux PRNG `/dev/random` and `/dev/urandom` security. In particular, we prove that these PRNGs are not robust. These properties are due to the behavior of the *entropy estimator* and the *mixing function* used to refresh its internal state. As pointed by Barak and Halevi [BH05], who advise against using run-time entropy estimation, we have shown vulnerabilities on the entropy estimator due to its use when data is transferred between pools in Linux PRNG. We therefore recommend that the functions of a PRNG do not rely on such an estimator.

Finally, we proposed a construction that meets our new property in the standard model and we showed that it is noticeably more efficient than the Linux PRNGs. We therefore recommend to use this construction whenever a PRNG with input is used for cryptography.

Acknowledgments: Yevgeniy Dodis' research was partially supported by gifts from VMware Labs and Google, and NSF grants 1319051, 1314568, 1065288, 1017471. Damien Vergnaud's research was supported in part by the French ANR-12-JS02-0004 ROMAnTIC Project. Daniel Wichs's research was partially supported by NSF grant 1314722.

References

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/.

[BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 05: 12th Conference on Computer and Communications Security*, pages 203–212, Alexandria, Virginia, USA, November 7–11, 2005. ACM Press.

[BK12] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A, 2012.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Berlin, Germany.

[CVE08] CVE-2008-0166. Common Vulnerabilities and Exposures, 2008.

[DGH+04] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 494–510, Santa Barbara, CA, USA, August 15–19, 2004. Springer, Berlin, Germany.

[DHY02] Anand Desai, Alejandro Hevia, and Yiqun Lisa Yin. A practice-oriented treatment of pseudorandom number generators. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 368–383, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Berlin, Germany.

[ESC05] D. Eastlake, J. Schiller, and S. Crocker. *RFC 4086 - Randomness Requirements for Security*, June 2005.

[GLSV12] François Goichon, Cédric Lauradoux, Guillaume Salagnac, and Thibaut Vuillemin. Entropy transfers in the Linux Random Number Generator. Rapport de recherche RR-8060, INRIA, September 2012.

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy*, pages 371–385, Berkeley, California, USA, May 21–24, 2006. IEEE Computer Society Press.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[ISO11] Information technology - Security techniques - Random bit generation. ISO/IEC18031:2011, 2011.

[Kil11] Killmann, W. and Schindler, W. A proposal for: Functionality classes for random number generators. AIS 20 / AIS31, 2011.

[Koo02] Philip Koopman. 32-bit cyclic redundancy codes for internet applications. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 459–472, Washington, DC, USA, 2002. IEEE Computer Society.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In Serge Vaudenay, editor, *Fast Software Encryption – FSE'98*, volume 1372 of *Lecture Notes in Computer Science*, pages 168–188, Paris, France, March 23–25, 1998. Springer, Berlin, Germany.

[LHA+12] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 626–642, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Berlin, Germany.

[LRSV12] Patrick Lacharme, Andrea Rock, Vincent Strubel, and Marion Videau. The linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012.

[NS02] Phong Q. Nguyen and Igor Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002.

[NZ96] Noam Nisan and David Zuckerman. Randomness is linear in space. *J. Comput. Syst. Sci.*, 52(1):43–52, 1996.

[Pol] PolarSSL is an open source and commercial SSL library licensed by Offspark B.V. https://polarssl.org.

[SHA95] Secure hash standard. National Institute of Standards and Technology, NIST FIPS PUB 180-1, U.S. Department of Commerce, April 1995.

[Sho06] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006.

[SV03] Amit Sahai and Salil P. Vadhan. A complete problem for statistical zero knowledge. *J. ACM*, 50(2):196–249, 2003.