Analysis of the Linux Pseudo-Random Number Generators

Yevgeniy Dodis
New York University
dodis@cs.nyu.edu

David Pointcheval
DI/ENS, ENS-CNRS-INRIA
david.pointcheval@ens.fr

Sylvain Ruhault
Oppida, France
ruhault@di.ens.fr

Damien Vergnaud
DI/ENS, ENS-CNRS-INRIA
vergnaud@diens.fr

Daniel Wichs
Northeastern University
wichs@ccs.neu.edu

Abstract

A pseudo-random number generator (PRNG) is a deterministic algorithm that produces numbers whose distribution is indistinguishable from uniform. A PRNG usually involves an internal state from which a cryptographic function outputs random-looking numbers. In 2005, Barak and Halevi proposed a formal security model for PRNGs *with input*, which involve an additional (potentially biased) external random input source that is used to refresh the internal state. In this work we extend the Barak-Halevi model with a stronger security property capturing how the PRNG should accumulate the entropy of the input source into the internal state after state compromise, even with a low-entropy input source—contrary to the Barak-Halevi model, which requires a high-entropy input source. More precisely, our new robustness 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. This expresses the real-life expected behavior of existing PRNG designs.

We show that neither the model nor the specific PRNG construction proposed by Barak and Halevi meets our robustness property, despite meeting their weaker robustness notion. On the practical side, we discuss the Linux `/dev/random` and `/dev/urandom` PRNGs and show attacks proving that they are not robust according to our definition, due to vulnerabilities in their entropy estimator and their internal mixing function.

Finally, we propose a simple PRNG construction that is provably robust in our new and stronger adversarial model. We therefore recommend the use of this construction whenever a PRNG with input is used for cryptography.

Keywords: randomness, entropy, security models, /dev/random

1. Introduction

Pseudo-Random Number Generators. Generating random numbers is an essential task in cryptography. Random numbers are necessary not only for generating cryptographic keys, but also in several steps of cryptographic algorithms or protocols (e.g., initialization vectors for symmetric encryption, password generation, nonce generation, etc.). 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 on the quality of the (pseudo-) randomness generated.

The lack of assurance about the generated random numbers can cause serious damage, and vulnerabilities can be exploited by attackers. One striking example is a failure in the Debian Linux distribution [4] that occurred when commented code in the OpenSSL PRNG with input led to insufficient entropy gathering and then to concrete attacks on the TLS and SSH protocols. More recently, Lenstra, Hughes, Augier, Bos, Kleinjung, and Wachter [16] showed that a nonnegligible percentage of RSA keys share prime factors. Heninger, Durumeric, Wustrow, and Halderman [10] presented an analysis of the behavior of Linux PRNGs that explains the generation of low-entropy keys when these keys are generated at boot time. Besides key generation cases, several works demonstrated that if nonces for the DSS signature algorithm are generated with a weak PRNG, then the secret key can be quickly recovered after a few key signatures are seen (see [17] and references therein). This illustrates the need for precise evaluation of PRNGs based on clear security requirements.

A user who has access to a truly random, possibly short, bit-string can use a *deterministic* (or *cryptographic*) PRNG to expand this short seed into a longer sequence whose 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 to refresh the internal state.

In spite of being widely deployed in practice, PRNGs with input were not formalized until 2005, by Barak and Halevi [1]. 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 analyzes the Linux `/dev/random` and `/dev/urandom` PRNGs.

Security Models. Descriptions of PRNGs with input are given in various standards [13, 11, 8]. They identify 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 [11, 13, 8, 2]. These standards consider adversaries with various means (and combinations of them): those who have access to the output of the generator; those who can (partially or totally) control the source of the generator; and those who can (partially or totally) control the internal state of the generator. Several requirements have been defined:

- **Resilience** – An adversary must not be able to predict future PRNG outputs even if the adversary can influence the entropy source used to initialize or refresh the internal state of the PRNG.
- **Forward security** – An adversary must not be able to identify past outputs even if the adversary can compromise the internal state of the PRNG.
- **Backward security** – An adversary must not be able to predict future outputs even if the adversary can compromise the internal state of the PRNG.

Desai, Hevia, and Yin [5] modeled a PRNG as an iterative algorithm and formalized the above requirements in this context. Barak and Halevi [1] model a PRNG with input as a pair of algorithms (`refresh, next`) and define a new security property called *robustness* that implies resilience, forward security, and backward security. This property assesses the behavior of a PRNG after compromise of its internal state and responds to the guidelines for developing PRNGs given by Kelsey, Schneier, Wagner, and Hall [12].

Linux PRNGs. 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 [1] discussed briefly the `/dev/random` PRNG, 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 [9]. It was completed in 2012 by Lacharme, Röck, Strubel, and Videau [15]. Gutterman et al. [9] presented an attack based on kernel version 2.6.10, for which a fix was published in the following versions. Lacharme et al. [15] give a detailed description of the operations of the PRNG and provide a result on the entropy preservation property of the mixing function used to refresh the internal state.

Our Contributions. On the theoretical side, we propose a new formal security model for PRNGs with input that encompasses all previous security notions. This new property captures how a PRNG with input should accumulate the entropy of the input data into the internal state, even if the former has low entropy only. This property was not initially formalized in [1], but it expresses the real-life expected behavior of a PRNG after a state compromise, where it is expected that the PRNG quickly recovers enough entropy, whatever the quality of the input.

On the practical side, we give a precise assessment of the two Linux PRNGs, `/dev/random` and `/dev/urandom`. We prove that these PRNGs are not robust and do not accumulate entropy properly, due to the behavior of their entropy estimator and their internal mixing function. We also analyze the PRNG with input proposed by Barak and Halevi [1]. This scheme was proven robust in [1], 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.

In this survey we give a high-level overview of our findings, leaving many lower-level details (including most proofs) to the conference version of this paper [7].

2. Preliminaries

Probabilities. When X is a distribution, or a random variable following this distribution, we denote $x \xleftarrow{\$} X$ when x is sampled according to X. The notation $X \leftarrow Y$ says that X is assigned the value of the variable Y , and that X is a random variable with a 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 by U_n.

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

$$\mathbf{H}_\infty(X) = \min_{x \in \mathrm{Supp}(X)} \{-\log \Pr[X = x]\}$$

where $\mathrm{Supp}(X) \subseteq S$ is the support of the distribution X.

Game Playing Framework. For our security definitions and proofs we use the code-based game playing framework of [3]. A game GAME has an $\mathsf{initialize}$ procedure, procedures to respond to adversary oracle queries, and a $\mathsf{finalize}$ procedure. A game GAME is executed with an adversary A as follows.

First, $\mathsf{initialize}$ executes, and its outputs are the inputs to A. Then A executes, its oracle queries being answered by the corresponding procedures of GAME. When 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 $\mathrm{GAME}^{\mathrm{A}} \Rightarrow y$ denote the event that this game output takes value y.

In the next section, for all $\mathrm{GAME} \in \{\mathrm{RES}, \mathrm{FWD}, \mathrm{BWD}, \mathrm{ROB}, \mathrm{SROB}\}$, $\mathrm{A}^{\mathrm{GAME}}$ denotes the output of the adversary. We let $\mathrm{Adv}_{\mathrm{A}}^{\mathrm{GAME}} = 2 \times \Pr[\mathrm{GAME}^{\mathrm{A}} \Rightarrow 1] - 1$. Our convention is that Boolean flags are assumed initialized to be false and that the running time of the adversary 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

Definition 1 (PRNG with Input). A PRNG with input is a triple of algorithm $\mathrm{G} = (\mathbf{setup}, \mathbf{refresh}, \mathbf{next})$ and a $(n,\ell,p)\in\mathrm{N}^3$ triple where

$\mathbf{setup}$ is a probabilistic algorithm that outputs some public parameters $\mathbf{seed}$ for the generator.

$\mathbf{refresh}$ is a deterministic algorithm that, given $\mathbf{seed}$, a state $S\in\{0,1\}^n$, and an input $I\in\{0,1\}^p$ outputs a new state $S' = \mathrm{refresh}(S,I) = \mathrm{refresh}(\mathrm{seed},S,I)\in\{0,1\}^n$

$\mathbf{next}$ is a deterministic algorithm that, given $\mathbf{seed}$ and a state $S\in\{0,1\}^n$, outputs a pair $(S',R) = \mathrm{next}(S) = \mathrm{next}(\mathrm{seed},S)$ 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 G.

Before defining our security notions, we notice that there are two adversarial entities that we need to worry about: the *adversary* A, whose task is (intuitively) to distinguish the outputs of the PRNG from random, and the *distribution sampler* D, whose task is to produce inputs $I_1, I_2, \ldots$, which have high entropy *collectively*, but somehow help A in breaking the security of the PRNG. In other words, the distribution sampler models a 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 sample 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 D.

$I\in\{0,1\}^p$ is the next input for the *refresh* algorithm.

γ is some *fresh entropy estimation* of I, as discussed below.

z is the leakage about I, given to the attacker A.

We denote by q_D the upper bound on the number of executions of D in our security games, and say that D is *legitimate* if[1]

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

for all $j\in\{1,\ldots,q_\mathrm{D}\}$ where $(\sigma_i, I_i, \gamma_i, z_i) = \mathrm{D}(\sigma_{i-1})$ for $i\in\{1,\ldots,q_\mathrm{D}\}$ and $\sigma_0 = 0$.

We now explain the reason for explicitly requiring D to output the entropy estimate γ_j used in (1). Most complex PRNGs, including the Linux PRNGs, are concerned with the situation in which the system might enter a prolonged state during which 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 [19]. This means that if we require our PRNG G to explicitly come up with such a procedure E, we are bound to either place some significant restrictions (or assumptions) on D , or rely on some ad hoc and nonstandard assumptions. Indeed, as part of this work we will demonstrate some attacks on the entropy estimation of the Linux PRNGs, illustrating how hard (or, perhaps, impossible) it is to design a sound entropy estimation procedure E. Finally, we observe that the design of E is anyway completely *independent* of the mathematics of the actual `refresh` and `next` procedures, meaning that the latter can and *should* be evaluated independently of the "accuracy" of E.

Motivated by these considerations, we do not insist on any "entropy estimation" procedure as a mandatory part of the PRNG design, which allows 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 D *itself*, which allows us to concentrate on the *provable* security of the `refresh` and `next` procedures. In particular, in our security definition we will not attempt to verify if D's claims are accurate (as we said, this appears hopeless without some kind of heuristics), but will only require security when D is *legitimate*, as defined in (1). Equivalently, we can think that the entropy estimations γ_j come from the entropy estimation procedure E (which is now "merged" with 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 estimate 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 D); b) we do not insist that a legitimate D can perfectly estimate the fresh entropy of its next sample I_j but only provide a *lower bound* γ_j that D is "comfortable" with. For example, 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 A via the leakage z_j![2] More generally, we allow 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 D gave us "no reason" to expect any security).

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 last being the strongest notion among them. We now define the analogs of these notions in our stronger adversarial model. Each of the games below is parameterized by some parameter γ^* (since which is part of the claimed PRNG security, and intuitively measures the minimal "fresh" entropy in the system when security is expected. In particular, minimizing the value of γ^* corresponds to a stronger security guarantee.

All four security games $(\mathrm{RES}(\gamma^*)$, $(\mathrm{FWD}(\gamma^*)$, $(\mathrm{BWD}(\gamma^*)$, $(\mathrm{ROB}(\gamma^*)$, are described using the game playing framework discussed earlier, and they share the same $\mathbf{initialize}$ and $\mathbf{finalize}$ procedures in Table 1.

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

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

As we mentioned, our overall adversary is modeled via a pair of adversaries (A, D) where A is the actual attacker and D is a stateful distribution sampler. We already discussed the distribution sampler D, so we turn to the attacker A, whose goal is to guess the correct value b picked in the **initialize** procedure, which also returns to A the public value **seed** and initializes several important variables: corruption flag **corrupt**, "fresh entropy counter" c, state S, and sampler's D initial state σ.[4] In each of the games (RES, FWD, BWD, ROB) A has access to the several oracles depicted in Table 2. We briefly discuss these oracles:

proc. initialize	**proc.** finalize (b^*)
seed $\overset{\$}{\leftarrow}$ setup ;	IF $b=b^*$ *RETURN* 1
$\sigma \leftarrow 0$; $S \overset{\$}{\leftarrow} \{0,1\}^n$; $c \leftarrow n$; corrupt $\leftarrow$ false ; $b \overset{\$}{\leftarrow} \{0,1\}$	ELSE *RETURN* 0
OUTPUT seed	

Table 1. The initialize and finalize procedures for **G**=(setup,refresh,next)

proc.D – refresh	**proc.** next – ror	**proc.** get – next	**proc.** get – state
$(\sigma, I, \gamma, z) \overset{\$}{\leftarrow} \mathsf{D}(\sigma)$	$(S, R_0) \leftarrow \text{next}(S)$	$(S, R) \leftarrow \text{next}(S)$	$c \leftarrow 0$, corrupt $\leftarrow$ true
$S \leftarrow \text{refresh}(S, I)$	$R_1 \overset{\$}{\leftarrow} \{0,1\}^\ell$	IF corrupt=true,	OUTPUT S
$c \leftarrow c + \gamma$	If corrupt=true,	$c \leftarrow 0$	
IF $c \geq \gamma^*$	$c \leftarrow 0$	OUTPUT R	**proc.** set–state (S^*)
corrupt $\leftarrow$ false	RETURN R_0		$c \leftarrow 0$, corrupt $\leftarrow$ true
OUTPUT (γ, z)	ELSE OUTPUT R_b		$S \leftarrow S^*$

Table 2. Procedures in games RES(γ^*), FWD(γ^*), BWD(γ^*), and ROB(γ^*), for **G**=(setup,refresh,next)

D–refresh. This is the key procedure in which the distribution sampler D is run, and whose 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 A. We denote by q_{D} the number of times A calls D–refresh (and hence D), and notice that by our convention (of including oracle calls into run-time calculations) the total run-time of D is implicitly upper bounded by the run-time of A.

next–ror/get–next. These procedures provide A calls with either the real-or-random challenge (provided corrupt=false) or the true PRNG output. As a small subtlety, a "premature" call to get–next before corrupt=false resets the counter c to 0, because then A might learn something nontrivial 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 give A the ability either to learn the current state S or to 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.

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

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

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

Definition 2 (Security of PRNG with Input). A PRNG with input G=(setup,refresh,next) is called $(T = (t, q_{\mathsf{D}}, q_R, q_S), \gamma^*, \varepsilon)$ *-robust (resp. resilient, forward-secure, backward-secure)* if, for any adversary A running in time at most t making at most q_{D} calls to D–refresh, q_R calls to next–ror/get–next and q_S calls to get–state/set–state, and any legitimate distribution sampler D inside the D–refresh procedure, the advantage of A in game ROB(γ^*) (resp. RES(γ^*), FWD(γ^*), BWD(γ^*)), is at most ε, where

ROB(γ^*) is the unrestricted game where A is allowed to make the above calls.

RES(γ^*) is the unrestricted game where A makes no calls to get–state/set–state (i.e., $q_S = 0$).

FWD(γ^*) is the restricted game where A makes no calls to set–state and a single call to get–state (i.e., $q_S = 1$), which is the last call that A is allowed to make.

BWD(γ^*) is the restricted game where A makes no calls to get–state and a single call to set–state (i.e., $q_S = 1$), which is the first oracle call that A is allowed to make.

Intuitively,

- Resilience protects the security of the PRNG when not corrupted against arbitrary distribution samplers D.
- Forward security protects past PRNG outputs if the state S is compromised.
- Backward security ensures that the PRNG can successfully recover from state compromise, provided enough fresh entropy is injected into the system.
- Robustness ensures arbitrary combinations of resilience, forward security, and backward security.

Hence, robustness is the strongest and the resilience is the weakest of the above four notions. In particular, all of 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 attributable to Barak and Halevi [1], 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 $\text{Extract}: \{0,1\}^p \to \{0,1\}^n$ and a standard deterministic PRG $\mathbf{G}: \{0,1\}^n \to \{0,1\}^{n+\ell}$. The modeling of [1] did not have an explicit **setup** algorithm, and the **refresh** and **next** algorithms are

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

$$\text{next}(S) = \mathbf{G}(S)$$

$\mathbf{G}'$ denotes the truncation of $\mathbf{G}$ to the first n output bits. However, we will also consider the "simplified BH" construction, wherein $\mathbf{G}'$ is simply the identity function (i.e., $\text{refresh}(S, I) = S \oplus \text{Extract}(I)$), since this variant will help us illustrate our attacks better and is already secure in a variant of the BH model that relaxes the strict requirement of "state pseudorandomness at all times" (while keeping the pseudorandomness of all outputs, which is the main property one cares about)

Attack on Simplified BH. Consider the following very simple distribution sampler D. At any time period, it simply sets $I = \alpha^p$ for a fresh and random bit α and also sets entropy estimate $\gamma = 1$ and leakage $z = \emptyset$. Clearly, D is legitimate. Hence, for any entropy threshold γ^*, the simplified BH construction must regain security after γ^* calls to the D–**refresh** procedure following a state compromise. Now consider the following simple attacker A attacking the backward security (and thus robustness) of the simplified BH construction. It calls $\text{set-state}(0^n)$, and then makes γ^* calls to D–**refresh** followed by many calls to **next–ror**. Let us denote the value of the state S after j calls to D–**refresh** by S_j and let $Y(0) = \text{Extract}(0^p)$, $Y(1) = \text{Extract}(1^p)$. Then, recalling that $\text{refresh}(S, I) = S \oplus \text{Extract}(I)$ and $S_0 = 0^n$ we see that $S_j = Y(\alpha_1) \oplus \ldots Y(\alpha_j)$, 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 D, the **refresh** procedure failed to accumulate more than 1 bit of entropy in the final state $S^* = S_{\gamma^*}$. In particular, after γ^* calls to D–**refresh**, 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 **next–ror**).

This shows that the simplified BH construction is *never* backward secure, despite being robust (modulo state pseudorandomness) in the model of [1].

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 **Extract** and PRG **G**. For **Extract**, we simply take any good extractor (possibly seeded) where $\text{Extract}(0^p) = \text{Extract}(1^p) = 0^n$. Such an extractor exists, since we can take any other initial extractor **Extract**, and simply modify it on inputs **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 **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 **Extract** and **G**, we can keep the same distribution sampler D and the attacker 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 $S' = \mathbf{G}'(S \oplus \text{Extract}(I)) = \mathbf{G}'(0^n \oplus 0^n) = 0^n = S$. 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 an **Extract** and **G**, much like in the simplified BH case. This turns out to be an interesting and rather nontrivial question, which relates to the randomness extraction properties (or lack of thereof) of the "CBC-MAC" construction (considered by [6] under some idealized assumptions about $\mathbf{G}'$).

Instead of following this direction, below we give an almost equally simple construction that is *provably robust* in the standard model, without any idealized assumptions.

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:

$\text{setup}()$: Output $\text{seed} = (X, X') \leftarrow \{0,1\}^{2n}$. $S' = \text{refresh}(S, I)$: Given $\text{seed} = (X, X')$, 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 F_{2^n}. $(S', R) = \text{next}(S)$: Given $\text{seed} = (X, X')$ 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 **refresh** procedure iteratively.

Theorem 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 G=(**setup,refresh,next**) be defined as above. Then G is a $((t', q_{\mathsf{D}}, q_R, q_S), \gamma^*, \varepsilon)$ -robust PRNG with input where $t' \approx t$, $\varepsilon = q_R(2\varepsilon_{prg} + q_{\mathsf{D}}^2 \varepsilon_{ext} + 2^{-n+1})$ as long as $\gamma^* \geq m + 2\log(1/\varepsilon_{ext}) + 1, n \geq m + 2\log(1/\varepsilon_{ext}) + \log(q_{\mathsf{D}}) + 1$.

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 are used in the OS security services and some cryptographic libraries. We give a precise description[6] of them in our model as a triple **LINUX**=(**setup,refresh,next**) and we prove the following theorem:

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

Since the actual generator **LINUX** does not define any seed (i.e., the algorithm **setup** always outputs Ø), as mentioned above, it cannot achieve the notion of robustness. However, we additionally mount concrete attacks that would work even if **LINUX** had used a seed. 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. PRNG Overview

Security Parameters. The **LINUX** PRNG uses the parameters n=6144, ℓ=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 ℓ=80, bits and truncates the last block if necessary.

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

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.

Functions $\mathbf{refresh}$ and $\mathbf{next}$. There are two $\mathbf{refresh}$ functions: $\mathbf{refresh}_i$ that initializes the internal state and $\mathbf{refresh}_c$ that updates it continuously. There are two $\mathbf{next}$ functions: $\mathbf{next}_u$ `/dev/urandom` and $\mathbf{next}_r$ for `/dev/random`.

Mixing Function. The PRNG uses a *mixing function* $\mathbf{M}$ to mix new data in the input pool and to transfer data between the pools.

Entropy Estimator. The PRNG uses an *entropy estimator* that estimates the entropy of new inputs and the entropy of the pools. The PRNG uses these estimations to control the transfers between the pools and how new input is collected. This is described in detail in Section 5.2. The estimations are named E_i (entropy estimation of S_i), E_u (of S_u), E_r (of S_r).

5.2. Attacks Overview

Overview of the Attack on the Entropy Estimator. The PRNG uses an *entropy estimator* on each input that continuously refreshes the internal state of the PRNG. This estimator can be fooled in two ways. First, it is possible to define a distribution of zero entropy that the estimator will estimate to be of high entropy; second, it is possible to define a distribution of arbitrary high entropy that the estimator will estimate to be 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. With these distributions, an attacker can control the transfer of data between the pools and force the generator not to use fresh inputs when generating data.

Overview of the Attack on the Mixing Function. The PRNG uses a *mixing function* $\mathbf{M}$ to mix new data in the input pool. It is possible to define a distribution of arbitrary high entropy for which the mixing function is completely counterproductive (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. With this distribution, an attacker can force the internal state of the PRNG to contain only one bit of entropy and therefore easily predict its output.

6. Conclusion

We have proposed a new property for PRNG with input that captures how it should accumulate entropy into the internal state. This property expresses the real expected behavior of a PRNG after a state compromise, when it is expected that the PRNG quickly recovers enough entropy, even with a low-entropy external input. We gave a precise assessment of the Linux `/dev/random` and `/dev/urandom` PRNGs. We proved that these PRNGs do not achieve this property, due to the behavior of their *entropy estimator* and their *mixing function*. As pointed out by Barak and Halevi [1], who advise against using run-time entropy estimation, our attacks are due to its use when data is transferred between pools in Linux PRNGs. 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. Thus, from the perspective of provable security, our construction appears to be vastly superior to Linux PRNGs. We therefore recommend the use of this construction whenever a PRNG with input is used for cryptography.

7. Acknowledgments

Yevgeniy Dodis's research was partially supported by the gift from VMware Labs and NSF grants 1319051, 1314568, 1065288, and 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.

8. References

1 Barak, B. and Halevi, S. A model and architecture for pseudo-random generation with applications to /dev/random. In *ACM CCS 05 12th Conference on Computer and Communications Security* (Nov. 2005), V. Atluri, C. Meadows, and A. Juels, Eds., ACM Press, pp. 203–212.

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

3 Bellare, M. and Rogaway, P. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT Advances in Cryptology – EUROCRYPT 2006* (May / June 2006), S. Vaudenay, Ed., vol. 4004 of LNCS Lecture Notes in Computer Science, Springer, pp. 409–426.

4 CVE-2008-0166. Common Vulnerabilities and Exposures, 2008.

5 Desai, A., Hevia, A., and Yin, Y. L. A practice-oriented treatment of pseudorandom number generators. In *EUROCRYPT Advances in Cryptology – EUROCRYPT 2002* (Apr. / May 2002), L. R. Knudsen, Ed., vol. 2332 of *LNCS Lecture Notes in Computer Science*, Springer, pp. 368–383.

6 Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., and Rabin, T. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In *CRYPTO Advances in Cryptology – CRYPTO 2004* (Aug. 2004), M. Franklin, Ed., vol. 3152 of *LNCS Lecture Notes in Computer Science*, Springer, pp. 494–510.

7 Dodis, Y., Pointcheval, D., Ruhault, S., Vergnaud, D., and Wichs, Daniel, Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *ACM Conference on Computer and Communication Security (CCS)*, November 2013.

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

9 Gutterman, Z., Pinkas, B., and Reinman, T. Analysis of the Linux random number generator. In *2006 IEEE Symposium on Security and Privacy* (May 2006), IEEE Computer Society Press, pp. 371–385.

10 Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium* (Aug. 2012).

11 Information technology - Security techniques - Random bit generation. ISO/IEC18031:2011, 2011.

12 Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Cryptanalytic attacks on pseudorandom number generators. In *FSE Fast Software Encryption – FSE'98* (Mar. 1998), S. Vaudenay, Ed., vol. 1372 of *LNCS Lecture Notes in Computer Science*, Springer, pp. 168–188.

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

14 Koopman, P. 32-bit cyclic redundancy codes for internet applications. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2002), DSN '02, IEEE Computer Society, pp. 459–472.

15 Lacharme, P., Rock, A., Strubel, V., and Videau, M. The Linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012.

16 Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Public keys. *CRYPTO Advances in Cryptology – CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 LNCS Lecture Notes in Computer Science, Springer, pp. 626–642.

17 Nguyen, P. Q. and Shparlinski, I. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology* 15, 3 (2002), 151–176.

18 Nisan, N. and Zuckerman, D. Randomness is linear in space. *J. Comput. Syst. Sci. 52*, 1 (1996), 43–52.

19 Sahai, A. and Vadhan, S. P. A complete problem for statistical zero knowledge. *J. ACM 50*, 2 (2003), 196–249.

20 Shoup, V. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006.