Pseudo-Random Generators in Practice

Nishant Kumar, Tapas Jain

April 30, 2015

Abstract

A pseudo random number generator (PRNG) is a deterministic algorithm that produces numbers whose distribution is indistinguishable from uniform. Such PRNG needs a short random seed to generate perfectly random numbers, but users donot have access to such random short seed. So, in practice PRNG with entropy inputs is used. Examples of such PRNGs are the Linux's PRNGs - /dev/random and /dev/urandom. There has been some work in the recent past to formalize the notion and security definitions of PRNG with inputs. In this report, we focus on these works and then present some new attacks and insights into the Linux's PRNG.

1 Introduction

Generating random numbers is integral to whole of cryptography. It is required in various applications and protocols like key generation, nonce generation, etc. Typically, cryptographers assume the users of such protocols have access to perfect randomness and prove the security of their schemes assuming this. But in practice, it is not possible for users to have access to perfect randomness. Rather they use, what is called a pseudo random number generator (PRNG). These PRNG are deterministic and when given a short random seed can expand this seed into a longer pseudo-random number. But, it is unrealistic to assume that users have access to such short random seed. In a PRNG with input, the users store a secret random state and have access to a (potentially biased) random source.

Linux's /dev/random, /dev/urandom; Mac OS's Yarrow and Microsoft Windows' Fortuna are examples of such PRNG with input. Linux's PRNG was first time incorporated into the linux kernel in 1994. But it was only in 2005 that Barak and Halevi ([1]) tried to formally model PRNG with inputs and propose corresponding security notions. Since then there has been much work to improve upon the model and provide better security guarantees.

The next section discusses some of the literature we found on the topic. The following sections illustrate the modelling of PRNG with inputs and

state the known security notions. Finally the last 2 sections contain the description of the Linux PRNG with known and new attacks and insights.

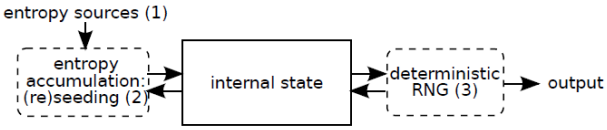

Figure 1: Model of PRNG with entropy inputs

2 Related Work

Past work on PRNG with inputs defined 3 security notions with different amount of powers being allowed to the adversary. These included

- Resiliance : 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.

In 2005, Barak and Halevi ([1]) formally modelled PRNG with inputs as a pair of algorithms - (refresh,next) and introduced a new security notion of robustness, that implied the previous 3 security notions. This property actually assesses the behavior of a PRNG after compromise of its internal state.

Further, in 2013, Dodis et al ([2]) improved upon the previous model and made the adversary more stronger by removing the burden of entropy estimation from the PRNG and allowing the adversary himself to specify the entropy to the PRNG. This model is what has been elaborated in this report. It is also worth mentioning here that this gives a lot of power to the adversary in the sense that he can tell the prng the entropy estimate and security after a state compromise is only expected when the adversary provides inputs of however low entropy s.t. the total entropy contained in the inputs crosses some threshold value.

Also, in 2014, [3] proposed and addressed the problem whereby after a state compromise the PRNG wants to block the output until sufficient entropy has gathered. But any premature production of output will lead to a total loss in the amount of entropy gathered till then. The main challenge in such scenarios is to block without knowing the timing of the last compromise. [3] formally models this problem and proposes a near optimal construction for

the same using ideas from the Fortuna RNG. We will not be elaborating on this problem in this report and will mainly try to focus on the Linux RNG after formally modelling it.

3 PRNG with Input : Modelling and Security

3.1 Model

Definition : PRNG with input consists of three algorithms $(setup, refresh, next)$ and takes three arguments $(n, l, p) \in \mathbf{N}^3$ where n is the length of the state, l is the length of the output generated by the PRNG and p is the length of the input from the random source.

- *setup* : It is a probabilistic algorithm and outputs some *public parameters seed* for the generator.
- $refresh$: It is a deterministic algorithm that, takes *seed*, a state $S \in \{0,1\}^n$ and an input $I \in \{0,1\}^p$ and outputs a new state $S' = refresh(seed, S, I) \in \{0,1\}^n$ which may have some enhanced entropy in the PRNG.
- *next* : It is also a detrministic algorithm that, takes *seed* and a state $S \in \{0,1\}^n$ outputs a pair $(S', R) = next(seed, S)$ where $S' \in \{0,1\}^n$ denotes the new state and $R \in \{0,1\}^l$ denotes the output of the PRNG.

3.2 Security

For the security notion, two adversarial entities are defined :- the attacker A who has to distinguish the outputs generated by the PRNG from perfectly random numbers with non-negligible probability and the distribution sampler D whose task is to produce the inputs $I_1, I_2, ...$ which are given to the PRNG, have high entropy collectively and (somehow) help the attacker A to break the security of PRNG.

3.2.1 Attacker

The attacker is assigned the task to distinguish between the two outputs one from the PRNG and one from a (theoretical) perfect random generator with a non-negligible probability using the help of the distribution sampler D.

3.2.2 Distribution Sampler

It is basically a probabilistic algorithm which, given its current state σ, outputs

- σ' new state for D.
- $I \in \{0,1\}^p$ next input for the PRNG algorithm.
- γ Some entropy estimation for I
- z leakage about I given to A.

Suppose if q_D is the maximum number of executions of D. Then D is legitimate if :

$$\begin{aligned} &\mathbf{H}_\infty(I_j|I_1, I_2, ..., I_{j-1}, I_{j+1}, .., I_{q_D}, z_1, z_2, ..., z_{q_D}, \gamma_1, \gamma_2, ..., \gamma_{q_D}) \geq \gamma_j \\ &\forall j \in \{1, 2, ..., q_D\} \ \ and \ \ (\sigma_i, I_i, \gamma_i, z_i) = D(\sigma_{i-1}) \end{aligned} \tag{1}$$

This is based on the assumption that at the start, all the outputs by the D can be obtained. Now this equation tells us that given all the previous and future outputs by the D, γ_j is the lower bound on the minimum entropy possible for input I_j, i.e. the distribution sampler is legitimate only if he gives the prng entropy estimates which are less than or equal to the actual values.
In this model, γ is been given by the adversary but is not the case in general PRNG's as are used in the real life, example Linux's PRNG or Windows PRNG. They have their inherent entropy estimators which estimates the entropy of the input. This is one of the stronger assumptions been made in the model, which increases the power of the adversary from that presented in Barak and Halevi([1]). Also the work of the entropy estimator is to block the PRNG from giving any output value until the state has an accumulated entropy say, $\gamma*$ which is some threshold value. The security definition described by [2] does not try to verify if the claims coming from D are legitimate or not, and builds on the assumption that they are legitimate and provide security for that.

3.3 Security Notions

Four security notions for PRNG with input have been defined : Resilience (RES), Forward (FWD), Backward (BWD) and Robustness (ROB). These security notions are described on the game as shown below :
The attacker's A goal here is to guess the correct value of b picked in the *initialize* procedure which is initializing several important variables : corruption flag *corrupt*, fresh entropy counter c, state S and $D's$ initial state σ.
Now to define the security notion, A has been given accesses to oracle described below :

Algorithm 1 proc.initialize

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

Algorithm 2 proc.finalize

if $b = b*$ **then return** 1
else return 0

Algorithm 3 proc.D-refresh

$(\sigma, I, \gamma, z) \xleftarrow{\$} D(\sigma)$
$S \leftarrow refresh(S, I)$
$c \leftarrow c + \gamma$
if $c \geq \gamma*$ **then**
    $corrupt \leftarrow false$
OUTPUT (γ, z)

Algorithm 4 proc.next-ror

$(S, R_0) \leftarrow next(S)$
$R_1 \xleftarrow{\$} \{0,1\}^l$
if $corrupt = true$ **then**
    $c \leftarrow 0$
    return R_0
else
    OUTPUT R_b

Algorithm 5 proc.get-next

$(S, R) \leftarrow next(S)$
if $corrupt = true$ **then**
    $c \leftarrow 0$
OUTPUT R

Algorithm 6 proc.get-state

$c \leftarrow 0$
$corrupt \leftarrow true$
OUTPUT S

Algorithm 7 proc.set-state(S^*)

1: $c \leftarrow 0$
2: $corrupt \leftarrow true$
3: $S \leftarrow S^*$

- $D - refresh$: In this procedure, the distribution sampler is run and its input is given to the PRNG to refresh its state. Also, the entropy counter is updated and if this updated value exceed the threshold value, corrupt is updated to false. A total number of q_D calls are made to this procedure.

- $next - ror/get - next$: These procedures provide the real or random challenges (depending on the value of b and provided corrupt=false) or the true PRNG output. Also, a premature calls to them resets the entropy counter to 0. q_R denotes the total number of calls to either of these. It is worth mentioning here that reseting the value of c to 0, means that we believe once the state is compromised, any premature production of output will lead to a total loss in entropy gathered till then.

- $get - state/set - state$: These procedures allow the A to learn the current state S or set it to S^*. Also, in both of these, c is set to 0 and corrupt to true. q_S denote the total number of calls to either of these.

For the convenience, the resources of A is denoted by $T = (t, q_D, q_R, q_S)$ where t is the running time of A.

Definition for Security of PRNG with Input : A PRNG generator with input is called $(T, \gamma*, \epsilon)$-robust(resp. resilient, forward-secure and backward-secure) if for any attacker A running in time t, making atmost 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 D inside $D - refresh$ procedure, the advantage of the A in game $ROB(\gamma*)$ (resp. $RES(\gamma*)$, $FWD(\gamma*)$, $BWD(\gamma*)$) is atmost ϵ, where:

- $ROB(\gamma*)$ unrestricted game where A is allowed to make all the above calls.

- $RES(\gamma*)$ restricted game where A makes no calls to $get - state/set - state$. It protects the security of the PRNG when not corrupted against arbitrary D.

- $FWD(\gamma*)$ restricted game where A makes no calls to $set - state$ and a single call to $get - state$ which is the last oracle call A is allowed to make. It protects past outputs if the current state is compromised.

- $BWD(\gamma*)$ restricted game where A makes no calls to $get-state$ and a single call to $set-state$ which is the very first oracle call it is allowed to make. It protects the future outputs if current state is compromised.

The Advantage of the A is defined as $|2 * Pr(b^* = b) - 1|$.

3.4 Simpler Notions of PRNG Security

[2] define two properties of PRNG with input which taken together implies robustness.

1. **Recovering Security**

   - It considers an atacker which has compromised the state to some arbitrary value S_0.
   - Then after that sufficiently many calls to $D-refresh$ were made to make the entropy of the the PRNG above the threshold value and corrupt flag to false. Suppose this results to some state S.
   - Now, the output from PRNG $(S^*, R) \leftarrow next(S)$ looks indistinguishable from random.

   We say that PRNG with input has $(t, q_D, \gamma^*, \epsilon)$-recovering security if for any attacker A and legitimate distribution sampler D both running in time t, the advantage of the adversary with parameters γ^* and q_D is atmost ϵ.

2. **Preserving Security**

   - The initial state of the PRNG is uncompromised and uniformly random.
   - Now this state is refreshed with arbitrary (adversarial) samples $I_1, I_2, ..I_d$. Suppose this results to some state S_d.
   - Now, the output from PRNG $(S^*, R) \leftarrow next(S_d)$ looks indistinguishable from random.

   A PRNG with input has (t, ϵ)-preserving security if the advantage of the attacker A running in time t is atmost ϵ.

Theorem : If a PRNG with input has both $(t', q_D, \gamma^*, \epsilon_r)$-recovering security and (t, ϵ_p)-preserving security, then it is $((t, q_D, q_R, q_S), \gamma^*, q_R(\epsilon_r + \epsilon_p))$-robust where $t' \approx t$.

4 Secure Construction of PRNG with input

[2] also describes a construction of the PRNG with input which is secure under the robustness defined by recovering and preserving security.
Let $G : \{0,1\}^m \rightarrow \{0,1\}^{n+l}$ be a (deterministic) pseudorandom generator where $m < n$. This construction takes n(state length), l(output length) and $p = n$(input length) as its parameters, and defines the three algorithms as follows :

- $setup()$: It outputs $seed = (X, X') \leftarrow \{0,1\}^{2n}$.
- $S' = refresh((X, X'), S, I)$: Given $seed, S$ *and* I, it updates the state S as $S' := S.X + I$.
- $(S', R) = next((X, X'), S)$: Given *seed and a state* S, it first compute $U = first\ m\ bits\ of\ (X'.S)$. Then it outputs $(S', R) = \mathbf{G}(U)$.

Some keypoints in this construction :

- It wasn't enough to just apply the first function,i.e, $S' := S.X + I$ and not take $U = first\ m\ bits\ of\ (X'.S)$. Just taking X was not enough. Suppose d calls for refresh were made, then new state $S' := S.X^d + I_{d-1}.X^{d-1} + ... + I_1.X + I_0$. In this, the probability of two distinct inputs to collide is $d/2^n$ which was not sufficiently universal to make it a good extractor and hence making it almost random. So it is important to also use an X' so that the function over I becomes a good randomness extractor. For further details, please refer the proof given in section 4 of [2].
- The proof of robustness can be done in two parts, including proving recovering security and preserving security. [2] uses a hybrid argument to prove these securities. The hardness assumption behind the proofs is Information Theoretic. Also, the recovering security uses the assumption that the distribution sampler D is legitimate.
- Taking the first m bits in the calculation of U was to ensure the state pseudo-randomness of the construction which is based on construction of [1]. If the first m bits were not truncated, then a very strong attack is possible on the construction.

5 Linux PRNG

The linux PRNG - *LINUX* consists of 2 PRNGs with input /dev/random and /dev/urandom. /dev/random is a blocking PRNG with input, while urandom is non-blocking. What this means is that if the PRNG runs low on entropy (as estimated by its entropy estimator), random blocks and does

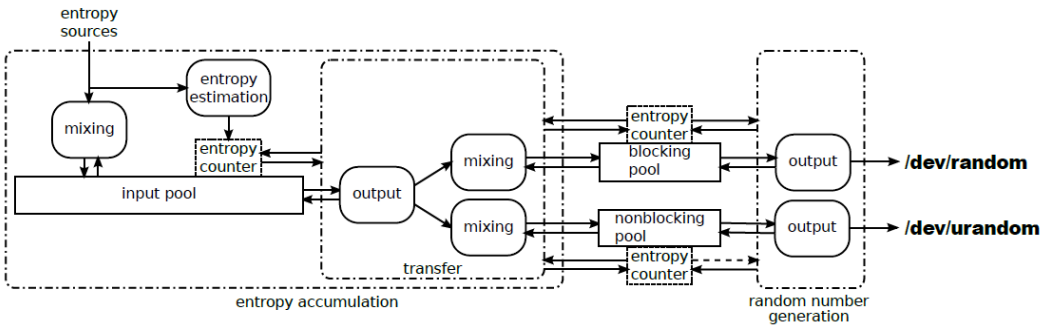

Figure 2: Linux PRNG overview

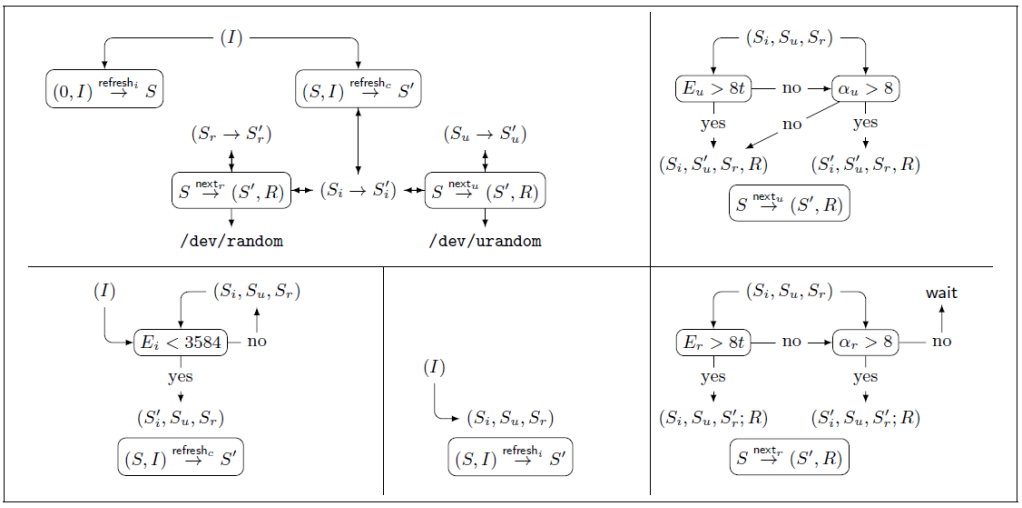

Figure 3: State and pool changes in Linux's PRNG

not produce further output until sufficient entropy has gathered back. On the other hand urandom continues to produce output even if it runs low on entropy. Also, its worth mentioning that these PRNGs are used in numerous OS security services and cryptographic libraries.
The following parts contain a precise description of the PRNG in the model described in the previous sections.

- *LINUX* contains state $S = (S_i, S_r, S_u)$ and has the 3 algorithms (setup, refresh, next). Also, $|S_i| = 4096$ bits, $|S_r| = 1024$ bits and $|S_u| = 1024$ bits. S_i is called the input pool and is where the new entropy is collected. S_r and S_u are the output pools used by /dev/random and /dev/urandom respectively. When the output pools contain sufficient amount of entropy, output is produced using them and if they run low, entropy is transferred from the input pool to the output pools

using the mixing function.

- $Setup \rightarrow seed$

  Let $seed \leftarrow \epsilon$ where ϵ is the empty string. Output $seed$.

- $Refresh(seed, S, I) \rightarrow S'$

  The refresh procedure of LINUX consists of two algorithms - one is $refresh_i$ and other is $refresh_c$ as described in algorithms 8 and 9. $refresh_i$ is called on initializing the internal state and $refresh_c$ is called on every other normal refresh operation. For details on what the inputs are and how they are collected, please refer section 5.2 of [2], and sections 2.2 and 3.1 of [4]. The important point to note is that if the PRNG estimates internal pool having sufficient entropy, (specifically if $E_i \geq 3584$) the new input is neglected. This will later form the basis of attacks on the PRNG. Note that M is the mixing function used to mix new input into the pool.

Algorithm 8 $Refresh_i$

Require: Inputs I_1 and I_2, $S = 0$
Ensure: $S = (S_i, S_r, S_u)$
1: $S_i \leftarrow M(I_1, 0)$
2: $S_r \leftarrow M(I_2, 0)$
3: $S_u \leftarrow 0$

Algorithm 9 $Refresh_c$

Require: Input $I, S = (S_i, S_r, S_u)$
Ensure: $S' = (S'_i, S'_r, S'_u)$
1: **if** $E_i \geq 3584$ **then**
2: $\quad S'_i = S_i$
3: **else**
4: $\quad e \leftarrow Ent(I)$
5: $\quad S'_i = M(I, S_i)$
6: $\quad E_i = e + E_i$
7: $(S'_r, S'_u) \leftarrow (S_r, S_u)$

- $next(seed, S) \rightarrow (S', R)$

  The next procedure is divided into 2 parts - one $next_r$ - for output from /dev/random and one $next_u$ for output from /dev/urandom. The $next_r$ procedure is described in algorithm 10. Basically $\alpha_r \geq 8$ means that input pool has sufficient entropy and then using the folding and hash functions - F, H data is transferred from the input pool to the output pool. Since in this report, we donot require the specific use

of $next_u$ algorithm, we donot state it formally here. Please refer to Algorithm 4 of [2] for details. But, basically the algorithm for $next_u$ is the same as for $next_r$, except that even if enough entropy is not available, i.e. if $\alpha_r < 8$ in the algorithm of $next_r$, $next_u$ produces output from S_u.

Further details and insights can be found in section 5.3 of [2] and section 3.3 of [4].

Also, the functions F, H used in the algorithms are the folding and the hash functions. The hash function used is the SHA-1 hash funcion. We not elaborate on these here as these of little consequence to the attacks explained later. Further details can be found in section 5.5 of [2].

Algorithm 10 $Next_r$

Require: $t, S = (S_i, S_r, S_u)$
Ensure: $R, S' = (S'_i, S'_r, S'_u)$
1: **if** $E_r \geq 8t$ **then return** $R \leftarrow F \circ H \circ M(S_r, H(S_r))$
2: $\alpha_r \leftarrow min(min(max(t, 8), 128), \lfloor E_i/8 \rfloor)$
3: **if** $\alpha_r \geq 8$ **then**
4: $\quad T_i \leftarrow F \circ H \circ M(S_i, H(S_i))$
5: $\quad S'_i \leftarrow M(S_i, H(S_i))$
6: $\quad S^*_r \leftarrow M(S_r, T_i)$
7: $\quad E_i \leftarrow E_i - 8\alpha_r$
8: $\quad E_r \leftarrow E_r + 8\alpha_r$
9: $\quad S'_r \leftarrow M(S^*_r, H(S^*_r))$
10: $\quad R \leftarrow F \circ H \circ M(S^*_r, H(S^*_r))$
11: $\quad E_r \leftarrow E_r - 8t$
12: **else**
13: $\quad$ Block until $\alpha_r \geq 8$
14: $S'_u \leftarrow S_u$ **return** R, S'

- The entropy estimator : $Ent(I) \rightarrow H_i$

  Understanding this constitutes an important prerequisite to the attack presented in the next sections. The important point to note is that the algorithm considers the jiffies count only, where jiffies is some measure of time, to predict entropy. The other parts of input are completely ignored. This makes it susceptible to being fooled. Specifically, the estimator will estimate highly regular inputs of high entropy to have an entropy of 0 and vice versa. We describe this in detail later.

- The mixing function : $M(S, I) \rightarrow S'$

  Basically, the mixing function is used by *LINUX* whenever it needs to mix new input data in the pools or whenever a transfer is required

Algorithm 11 Entropy Estimator

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

between the input and the output pools. We donot delve into the exact mixing function as this is not required for the attacks we elucidate in this report, but suggest interested readers to refer sections 5.6 of [2] and section 3.1 of [4].

6 Attacking the Linux PRNG

6.1 Distributions used in attacking the entropy estimator

LINUX uses an internal 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 of high entropy, and 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 the following lemmas.

Lemma 1 *There exists a stateful distribution D_0 such that $H_\infty(D_0) = 0$, whose estimated entropy by LINUX is high.*

Proof. On input the state i, D_0 updates its state to $i+1$ and outputs a triple $[W_1^i, W_2^i, W_3^i]$ which constitutes the 12 byte input to the linux prng and where

$$W_1^0 = 2^{12}, W_1^i = |\lfloor \cos(i) \cdot 2^{20} \rfloor| + W_1^{i-1}$$

$$W_2^i = W_3^i = 0$$

Here W_1^i plays the role of jiffies count in the input to the entropy estimator. Then $H_\infty(D_0) = 0$ (conditioned on the future and past outputs) but the estimated entropy by *LINUX* is $H_i = 11$ (since $\Delta_i > 2^{12}$). ■

Lemma 2 *There exists a stateful distribution D_1 such that $H_\infty(D_1) = 64$, whose estimated entropy by LINUX is 0.*

Proof. On input the state i, D_1 updates its state to $i + 1$ and outputs a triple $[W_1^i, W_2^i, W_3^i]$ which constitutes the 12 byte input to the linux prng and where

$$W_1^i = i, W_2^i = W_3^i = U_{32}$$

Then $H_\infty(D_1) = 64$ (since each 12 byte input contains 8 bytes on random data) but the estimated entropy by *LINUX* is $H_i = 0$ (since $\delta_i = 1, \delta_i^2 = 0, \delta_i^3 = 0$ and so $H_i = 0$). ■

6.2 Concrete attacks already in literature

In this subsection concrete attacks on the robustness of the prng are discussed. Note that these attacks are taken from [2].

Lemma 3 */dev/random and /dev/urandom are not robust.*

Attack on /dev/random
Consider an adversary aganist the robustness of the Linux PRNG in the game $ROB(\gamma^*)$. After the initialize procedure the adversary makes the following oracle queries.

- **get-state**: After this call, $\mathcal{A}$ knows all the parameters of the prng - i.e. he/she knows $S_i, S_r, S_u, E_i, E_r, E_u$ and entropy counter $c = 0$.
- **next-ror**: $\mathcal{A}$ makes $\lfloor E_i/10 \rfloor + \lfloor E_r/10 \rfloor$ queries to **next-ror** oracle. Each call to next-ror will produce an output from the prng and the entropy of input pool will keep on decreasing. At the end $\mathcal{A}$ knows S_i, S_r ,$E_i = E_r = 0$ and also $c = 0$.
- **$\mathcal{D}$-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 $J \leftarrow U_{128}$, the uniform distribution. As $E_i = 3584$, these inputs are ignored. After 30 queries, $\mathcal{A}$ knows S_i and S_r and $E_r = 0, c = 3840$.
- **next-ror**: Supposedly if c has crossed the threshold entropy γ^*, next-ror provides it with the challenge output from either the prng or from the uniform distribution. But, since $E_r = 0$, a transfer is necessary

between S_i and S_r before generating R from the prng. 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 from the prng.

Analysis The adversary always outputs guess $b = 0$ (i.e. from the PRNG) if his estimate for R matches the challenge value from **next-ror** and 0 otherwise. Note that the output from next-ror is by default 80 bit string which is from U_{80} if $b = 1$ and from $next(S)$ if $b = 0$. The advantage of the adversary in the above game is

$$Pr[b = b^*] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 - \frac{1}{2^{80}})$$

$$\Rightarrow |Pr[b = b^*] - \frac{1}{2}| = \frac{1}{2}(1 - \frac{1}{2^{80}})$$

which is clearly non-negligible.

Explanation and intuition The above attack is exactly as presented in [2]. But, we believe it is wrong on some minor details. For example, the number of queries in the step 2 of the attack should be $Q = \lceil E_i/80 \rceil + \lceil E_r/80 \rceil$. Each call to next-ror will call the next procedure of the PRNG and since the default output of the PRNG is for 80 bits, so it should take Q number of queries to make $E_i = E_r = 0$. The main point is that after some calculable number of queries, the adversary can be sure that $E_i = E_r = 0$.

Similarly, in step 3 of the attack, we believe the number of queries to $\mathcal{D}_0$ should be $\lceil 3584/11 \rceil = 326$. This is of minor consequence, as the main thing is that after certain number of queries to $\mathcal{D}_0$, $E_i >= 3584, E_r = 0$ and LINUX begins to ignore any further inputs.

Another point to note is that the second stage of the step 3, where the PRNG is refreshed with inputs from uniform distribution, is only to make the value of c, the entropy counter maintained by the oracle sufficient enough to cross γ^*. In view of this, we believe the number of queries made to $\mathcal{D} - refresh$ with input from U_{128} should be $\lceil \gamma^*/128 \rceil$. To mount other attacks, it can just be borne in mind to somehow make the states and the next output of PRNG predictable and later executing sufficient number of $\mathcal{D} - refresh$ queries to make c cross the threshold. As will be evident after seeing the other attacks, this step is almost used as a tool in every attack.

The first, second and last steps in the above attacks are common and are used as tools. For, example, after the second step, we have $E_i = E_r = c = 0$. The main idea lies in the third step where the intention is to create a state using which the next output of the PRNG becomes predictable. Here this is being achieved by using inputs from distribution D_0. *LINUX* estimates these inputs to have high entropy and correspondingly raises its estimate of E_i, but actually these have $H_\infty(D_0) = 0$. So, the attacker can keep track of the changes in the internal state of the PRNG.

Attack on /dev/urandom

Let us consider an adversary $\mathcal{A}$ against the robustness of the generator /dev/urandom in the game $ROB(\gamma^*)$ that makes the following oracle queries: one get-state that allows it to know S_i, S_u, E_i, E_u; $\lfloor E_i/10 \rfloor + \lfloor E_r/10 \rfloor$ queries to **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.

Explanation and intuition The above attack uses the inputs from the distribution D_1, which LINUX estimates of 0 entropy. But because the actual min entropy of inputs from D_1 is 64, the distribution sampler is legitimate in giving $\gamma_i = 64$. This causes the value of c to continue increasing, while maintaining LINUX with $E_i = E_u = 0$. So, after sufficient number of queries, value of c will cross the threshold γ^* and next-ror will return a challenge value R_b. But, $\mathcal{A}$ can predict this output because it knows $E_i = E_u = 0$ and also the values of S_i and S_u.

Another Attack on /dev/random and /dev/urandom

Consider an adversary against the robustness of the Linux PRNG in the game $ROB(\gamma^*)$. After the initialize procedure the adversary makes the following oracle queries.

- **get-state**: After this call, $\mathcal{A}$ knows all the parameters of the PRNG - i.e. he/she knows $S_i, S_r, S_u, E_i, E_r, E_u$ and entropy counter $c = 0$.
- **next-ror**: $\mathcal{A}$ makes $\lfloor E_i/10 \rfloor + \lfloor E_r/10 \rfloor$ queries to **next-ror** oracle. At the end $\mathcal{A}$ knows S_i, S_r, $E_i = E_r = 0$ and also $c = 0$.
- $\mathcal{D}$**-refresh**: $\mathcal{A}$ refreshes the PRNG with inputs from D_0. After 300 queries, $E_i = 3584, E_r = 0$. $\mathcal{A}$ knows $S_i, S_r, E_i, E_r = 0$ and $c = 0$.
- **next-ror with t=1** Since $E_r = 0$, a transfer is necessary from the input pool to the output pool before generating the output. Since $\alpha_r = min(min(max(t, 8), 128), \lfloor E_i/8 \rfloor) = min(8, \lfloor E_i/8 \rfloor) = 8$, new value of $E_i = E_i - 8\alpha_r = 3584 - 64 = 3520$ and $E_r = E_r + 8\alpha_r - 8t = 56$.
- **next-ror with t=7** Since the output pool contains sufficient entropy, no transfer is necessary and new $E_r = 0$.
- Repeat the 2 previous steps until $E_i = 0$ and only do one next-ror query with t=1 in the last step, so that at end $E_i = 0, E_r = 56$.
- Refresh from U_{128}. After $\lceil \gamma^*/128 \rceil$ queries, adversary knows S_r and $c >= \gamma^*$.

- **next-ror with t=7**: Since $E_r = 56$, no transfer is necessary between S_i and S_r before generating R. But as $\mathcal{A}$ knows S_r, then $\mathcal{A}$ knows R.

The advantage of the adversary in the above game is

$$Pr[b = b^*] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 - \frac{1}{2^{56}})$$

.
With probability 0.5 either b=0 and the adversary wins with probability 1 or with probability 0.5 b=1 and adversary wins with probability $1 - (1/2^{56})$.
Explanation

So, basically in this setting the adversary uses sample inputs from the distribution D_0 to get LINUX in a state where it estimates its input pool's entropy to be high, but since the data used in refresh had actual min entropy 0, the final state is completely predictable. Then using cleverly 1 byte outputs from the PRNG, the adversary forces LINUX prng to estimate the output pool's entropy to be 56. Since all the state is now known to the adversary, the next output is predictable and leads to adversary winning the security game with non-negligible probability.

6.3 New attacks on the Linux PRNG

In this subsection, we present our own insights and intuitions of the attacks used against the Linux PRNG. We also present some new attacks on the PRNG, which although use the same basic tools as the previous attacks, are new in a sense that they have not been given elsewhere in literature.

- First of all, we note the attacks elucidated in the previous sections attack the robustness of the PRNG. We believe that any attempt to break the resiliance of the PRNG will fail from the tools used in the above attack. This is because the attacker needs some way to access the internal state of the PRNG to actually say something non-trivial about the output. If he/she is just allowed to make refresh queries and not make any get-state/set-state queries, the attacker will not be able predict the next output, atleast given only the distributions D_0 and D_1. So, we believe any attempts to break the resiliance of the PRNG is futile with these tools.

- Next, we move onto the forward and backward security of the PRNG. If thought carefully, the attacks presented in last sub-section seem to break the backward security of the PRNG, in the sense that once the attacker gets access to the state in present, he/she can predict the outputs in a later point in time. But the point remains that the model we defined in the previous sections, to break the backward security of the PRNG, the attacker must make the very first oracle query to

set-state. In the attacks presented in last sub-section, the very first oracle query is to get-state. The model defined does not allow set-state to get the attacker the current state, which should be possible intuitively. If the very first oracle query to get-state is replaced with one to set-state, we believe it leads to no change in the whole attack. Hence, in some sense, the attacks do break the backward security of the scheme.

- Another point we mention here in passing, is that the minor details of the attacks seem to mismatch the exact values. We already described this point earlier, when presenting the first of the attacks.

- The basic attacks presented can be modified and new attacks developed.

    1. In the second attack presented on /dev/random, the attacker is required to loop in next-ror with t=1 and t=7. We believe this is unnecessary. The attacker first gets S_i, S_r and executes step 2 of the attack (calling next-ror) to make $E_i = E_r = c = 0$. He then executes next-ror once with t=1. This makes $E_r = 56$. He can then increase c by refreshing input pool with data from either uniform distribution or D_1. Either way the only objective is to make c cross the threshold value γ^*, while not caring for E_i. The attacker then makes a query to next-ror with t=7, but since sufficient entropy is available in the output pool ($E_r = 56$), c has crossed its threshold value and he knows S_r, the next output will be generated from S_r and thus makes it predictable to the attacker.
    2. When refreshing the prng with the aim to increase the value of c, instead of using inputs from U_{128}, one can use inputs from D_1.
    3. Till now whatever we have discussed are in some sense modification of the original attacks. In an attempt to find a completely new attack, this is what we have thought out.

       As already discussed attacking the resiliance seems futile. So, the adversary should do a get-state/set-state as the first oracle query. This gets him the complete control of all the parameters and state in the prng. His aim then should be to somehow ensuring a predictable state, make some oracle queries s.t. if the prng is queried for the next prng output, it produces the value and since the state was known to the adversary, he will get to know the output. There are only 2 ways we believe this can be achieved. Either he ensures there is enough entropy in the output pool - S_r so that the next output is produced using only S_r or the entropy in S_r is 0 and S_i is known to him. For the latter of the two

it is also essential he somehow ensures that the input pool has sufficient entropy or else the prng will block. The latter of two approaches is what is done in the first attack presented in the last section whereby using inputs from D_0, he ensures that $E_r = 0$ and $E_i \geq 3584$ and also S_r, S_i are known. The former of the two is used by the 2nd attack presented in the last section, whereby the adversary ensures at the end $E_r = 56$ and does not care for E_i.

The next step of the attack is to somehow maintaining predictability, increase the value of c held by the oracles/environment. The attacks in [2] use a uniform distribution to achieve this, although other distributions like D_1 can be used. Then when the final next-ror is called, since c has crossed the threshold value, the procedure produces a challenge value. But since the adversary has ensured predictability, he can tell apart the real and ideal worlds with non-negligible probability.

- Based on the above discussion, we can present many new attacks, but all of them will be some minor modification of the above generic attack structure. So, in some sense we have not been able to find a completely new out-of-the-box attack, but identified a generic attack structure.

- An open question still remains as to how to exploit these theoretical attacks on the prng into actual real world practical attacks.

- All the previous analysis was done on Linux kernel version number 3.7.8 and the current stable version of Linux kernel is 4.0. Whether there has been further changes in the prng based on Dodis et al paper and the attacks presented is subject to source code analysis.

7 Conclusion

In this report we have tried to survey the recent literature on PRNG with inputs. Although, our attention has been centered around linux prng, several works similar to this have been done about other prng, specifically [5] is about proving the intel prng secure in such a model.
All the above discussion definitely points to the non-robustness of the Linux's prng. But these attacks are theoretical in the sense that they assume the attacker to get the full internal state of the prng. In practice, this means bypassing the linux kernel's security measures to get access to the kernel state, which is itself daunting task.
We mention here that [2] also contains attacks based on the mixing function used in the Linux PRNG. The attack is based on the linearity used by the mixing function and existentially proves the existence of a distribution such even after arbitrary number of refreshes from this distribution, the entropy

of the internal state does not increase. We have not touched upon this attack as our focus was based on finding new ways to exploit the entropy estimator.

References

1. Barak, B., Halevi, S.: A model and architecture for pseudo-random generation with applications to /dev/random. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, pp. 203–212. ACM, New York (2005)

2. Dodis, Y., Pointcheval, D., Ruhault, S., Vergniaud, D., Wichs, D.: Security analysis of pseudo-random number generators with input: /dev/random is not robust. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security, CCS 2013, pp. 647–658. ACM, New York (2013)

3. Dodis, Y., Shamir, A., Stephens-Davidowitz, N., Wichs, D.: How to eat your entropy and have it too – optimal recovery strategies for compromised rngs. Cryptology ePrint Archive, Report 2014/167 (2014), http://eprint.iacr.org/

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

5. A Provable-Security analysis of Intel's Secure PRNG :

   www.link.springer.com/chapter/10.1007/978-3-662-46800-5_4