





# Provable Security of Linux-DRBG in the Seedless Robustness Model

Woohyuk Chung<sup>1</sup>, Hwigyeom Kim<sup>2</sup><sup>\*</sup>, Jooyoung Lee<sup>1</sup><sup>\*\*</sup>, and Yeongmin Lee<sup>3</sup><sup>\*</sup>

<sup>1</sup> KAIST, Daejeon, Korea  
{hephaistus,hicalf}@kaist.ac.kr

<sup>2</sup> Norma Inc., Seoul, Korea  
rlagnlruea4@gmail.com

<sup>3</sup> DESILO Inc., Seoul, Korea  
yeongmin.lee@desilo.ai

**Abstract.** This paper studies the provable security of the deterministic random bit generator (DRBG) utilized in Linux 6.4.8, marking the first analysis of Linux-DRBG from a provable security perspective since its substantial structural changes in Linux 4 and Linux 5.17. Specifically, we prove its security up to  $O(\min\{2^{\frac{n}{2}}, 2^{\frac{\lambda}{2}}\})$  queries in the seedless robustness model, where  $n$  is the output size of the internal primitives and  $\lambda$  is the min-entropy of the entropy source. Our result implies 128-bit security given  $n = 256$  and  $\lambda = 256$  for Linux-DRBG. We also present two distinguishing attacks using  $O(2^{\frac{n}{2}})$  and  $O(2^{\frac{\lambda}{2}})$  queries, respectively, proving the tightness of our security bound.

**Keywords:** Deterministic random bit generator, Linux-DRBG, Seedless robustness, Provable security

## 1 Introduction

DETERMINISTIC RANDOM BIT GENERATOR. Producing random numbers plays a crucial role in cryptography, serving for the generation of secret keys, IVs and nonces (for encryption modes), and passwords (for identification protocols), to name a few. In practice, random bits are often generated using a deterministic random bit generator (DRBG), which refers to an algorithm that generates random bits using a seed value obtained from a physical source with a sufficient amount of entropy. The term “deterministic” means that there is no inherent randomness in the algorithm itself. DRBGs find applications in various environments such as simulation, encryption, etc.

<sup>\*</sup> This work was done while H. Kim and Y. Lee were PhD students at KAIST.

<sup>\*\*</sup> Jooyoung Lee was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) [NO.2022-0-01047, Development of statistical analysis algorithm and module using homomorphic encryption based on real number operation].

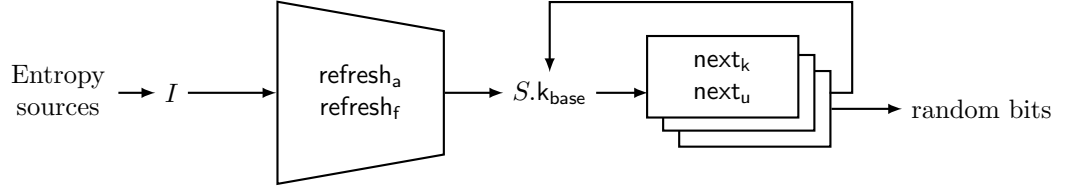


Fig. 1: Overall structure of Linux-DRBG.  $\text{refresh}_a$  and  $\text{refresh}_f$  are entropy absorbing functions,  $\text{next}$  and  $\text{next\_user}$  are random bit generating functions.

**PROVABLE SECURITY OF DRBG.** The randomness of the bits produced by a DRBG has typically been evaluated by statistical criteria. On the other hand, there have been attempts to prove the security of DRBGs through the provable security method in cryptography [4,17,5,13,19,22] as many constructions are based on cryptographic primitives such as block ciphers and hash functions. As part of this effort, Dodis et al. [9] proposed a security model for DRBGs, demonstrated that a random bit generator used in the Linux operating system is not secure under the proposed model, and suggested its modification. In this paper, the security notions for DRBGs are distinguished as robustness, forward security, backward security, and resilience. The security model incorporates a hardware random bit generator used to generate seeds in DRBGs and a virtual system distribution sampler to model the hardware random bit generator and adversarial manipulation on it. Based on this model, the robustness of the sponge structure has been proved [10], and subsequently, the robustness of HMAC-DRBG and HASH-DRBG, both recommended by NIST.SP.800-90A, was proved [20]. Recently, CTR-DRBG, also recommended by NIST.SP.800-90A, has been proved [14].

**SEEDLESS ROBUSTNESS MODEL.** Dodis et al. [7] pointed out a limitation of the existing model, which assumes and exploits randomness called a *seed*, unknown to adversaries and kept secret. The assumption is not realistic in a practical scenario, and a DRBG in such a model cannot be considered deterministic since the seed implies the existence of randomness in addition to entropy. They proposed a seedless robustness model and demonstrated that CTR-DRBG is not secure under the new model. They also proposed new DRBGs that are secure under the seedless robustness model.

**RESEARCH ON LINUX-DRBG.** Linux is one of the widely-used computer operating systems developed as open-source software through collaborative efforts within the community. Linux utilizes DRBGs to generate random bits. The incorporation of DRBGs in Linux dates back to version 1.3.30 in 1994, and since then, modifications and enhancements have been ongoing. Barak et al. [2] suggested the robustness model and discussed the robustness of Linux-DRBG, and Guttman et al. [12] introduced an attack on Linux 2.6.10 DRBG, and Linux has fixed the DRBG in following versions. Goichon et al. studied on entropy propagation in Linux-DRBG [11] in 2012. Dodis et al. [9] mentioned above, modified the

robustness model, demonstrating that Linux-DRBG is not robust exploiting its entropy estimating process with timer randomness. This vulnerability has been fixed in subsequent versions of Linux by modifying the Linux-DRBG to collect and estimate entropy from a variety of entropy sources. The Linux-DRBG has been significantly modified in Linux versions 4.0 and 5.17. However, the security of the updated Linux-DRBG has not yet been proven.

**THE STRUCTURE OF LINUX-DRBG.** The overall structure of the Linux-DRBG in version 6.4.8 of Linux is shown in Figure 1. Linux-DRBG collects and estimates entropy from a variety of entropy sources such as hardware, timers, interrupts, and bootloaders.

It then updates the base state with collected entropy, through entropy accumulating functions like procedure `refresha` and `refreshf`.

When a user runs random bit generating functions such as `nextk` and `nextu`, the state utilizes one of the CPU core’s states (CPU state) to update and then generate random bits. During this process, the base state is also re-updated.

Linux-DRBG utilizes two cryptographic primitives: for the entropy accumulating functions, `refresha` and `refreshf`, it uses the hash function BLAKE2s, and for the random bit generating functions, `nextk` and `nextu`, it employs the stream cipher ChaCha20. The internal structures of BLAKE2s and ChaCha20 have been modeled by Luykx et al.[15] and Degabriele et al.[8], respectively.

### 1.1 Our Contribution

Since the significant structural changes in Linux 4 and Linux 5.17, there has been no research on the provable security of Linux-DRBG. For the first time (to the best of our knowledge), we formally model the Linux-DRBG in Linux 6.4.8 and prove its security in the seedless robustness model.

According to the source code of Linux 6.4.8, Linux-DRBG has two entropy accumulating functions, `refresha` and `refreshf`, and two random bit generating functions, `nextk` and `nextu`. We abstracted Linux-DRBG into the 4 functions and adjusted its structure that does not fit into the existing seedless robustness model. The process of analyzing the source code to abstract Linux-DRBG is detailed in Section 4.

We prove that Linux-DRBG is secure up to  $O(\min\{2^{n/2}, 2^{\lambda/2}\})$  where  $n$  is the output size of the internal primitives and  $\lambda$  is the min-entropy of the entropy source. Since  $n = 256$  and  $\lambda = 256$  in Linux-DRBG, our security bound implies the 128-bit security of Linux-DRBG in the seedless robustness model. We also present two distinguishing attacks using  $O(2^{\frac{n}{2}})$  and  $O(2^{\frac{\lambda}{2}})$  queries, respectively, proving the tightness of our security bound.

Dodis et.al. used the reducing technique(called game hopping) to prove robustness by splitting the security into  $r$  individual recovering security and preserving security, where  $r$  is the number of random bit generating query [9]. Subsequent papers proving the robustness of DRBGs, except for cases like the direct proof of CTR-DRBG’s robustness in 2020 [14], have predominantly utilized this approach [10,20]. This methodology is also applied in the seedless robustness

model [7]. However, applying this method directly to Linux-DRBG would only yield  $n/3$ -bit security. In this paper, as shown in Lemma 3, we have adopted a different game hopping technique to split Linux-DRBG's robustness. Through this methodology, we could prove that Linux-DRBG is secure up to  $O(2^{n/2})$  adversarial queries. We believe that this proof method could be beneficial in proving the robustness of other DRBGs.

## 2 Preliminaries

We write  $0^n$  to denote the  $n$ -bit string of all zeros. Given a non-empty finite set  $\mathcal{X}$ ,  $x \leftarrow_{\$} \mathcal{X}$  denotes that  $x$  is chosen uniformly at random from  $\mathcal{X}$ . For a set  $\mathcal{X}$ ,  $|\mathcal{X}|$  denotes the number of elements in  $\mathcal{X}$ . The set of all permutations of  $\{0, 1\}^n$  is denoted  $\text{Perm}(n)$ . For a keyed function  $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$  with key space  $\mathcal{K}$  and non-empty sets  $\mathcal{X}$  and  $\mathcal{Y}$ , we will write  $F_K(\cdot)$  to denote  $F(K, \cdot)$  for  $K \in \mathcal{K}$ . Let  $S = \{a_1, \dots, a_s\}$ . Then, we write  $S \oplus x$  to denote  $\{a_1 \oplus x, \dots, a_s \oplus x\}$ . Let  $\chi = (A, B, C)$  be a list. We write  $\chi.\text{append}(D)$  to append an element to the  $\chi$ . Thus, after appending,  $\chi = (A, B, C, D)$ . We denote  $|$  as a bitwise OR operation.

For a (binary) string  $x$ ,  $|x|$  denotes the length of  $x$ . The empty string is denoted  $\varepsilon$ , where  $|\varepsilon| = 0$ . For an  $\ell$ -bit string  $x$ , and  $m$  and  $n$  such that  $0 \leq m \leq n \leq \ell - 1$ ,  $x[m : n]$  denotes an  $(n - m + 1)$ -bit string from the  $m$ -th bit to the  $n$ -th bit of  $x$ , and  $x[m : ]$  denotes an  $(\ell - m + 1)$ -bit string from the  $m$ -th bit to the last bit of  $x$ . When  $M = M_1 \parallel \dots \parallel M_w$  where  $|M_i| = t$  for  $1 \leq i \leq w - 1$  and  $0 < |M_w| \leq t$ , we write  $(M_1, \dots, M_w) \xleftarrow{t} M$ . For an integer  $0 \leq i < 2^s$ ,  $\langle i \rangle_s$  denotes  $s$ -bit representation of  $i$ . For a real number  $t$ ,  $\lceil t \rceil$  is the smallest integer that is the same as or bigger than  $t$ .

For a tuple  $S_A = (X, Y, Z)$ , we can access  $X$  inside  $S_A$  as  $S_A.X$ .

**RANDOM PERMUTATION.** A random permutation is a bijective mapping from a finite set to itself, selected uniformly at random from all possible permutations of the set. We treat 20 rounds of a ChaCha20 cipher as a random permutation  $\pi$  which is selected from  $\text{Perm}(2n)$  [3, 8].

**BLOCK CIPHERS.** A block cipher, modeled as an *ideal cipher*, is a keyed function  $E : \{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$  where for a fixed key  $K \in \{0, 1\}^k$ ,  $E_K(\cdot)$  is a random permutation that is uniformly chosen from  $\text{Perm}(n)$ . For the rest of the paper, we let  $\Pi(k, n)$  denote the set of all  $n$ -bit block ciphers using  $k$ -bit keys.

If a security proof supposes a block cipher is picked uniformly random from  $\Pi(k, n)$  at the beginning of the query and allows the adversary to make queries to the block cipher, we call the proof is modeled under an *ideal cipher model*.

**WEAK BLOCK CIPHER.** Consider a partition  $\{0, 1\}^n = \mathcal{W} \cup (\{0, 1\}^n \setminus \mathcal{W})$ . Define the set of weakly ideal ciphers  $\Pi_w(k, n, \mathcal{K}, \mathcal{W})$  with a weak key set  $\mathcal{K}$  as the collection of all  $E \in \Pi(k, n)$  that satisfies the following properties: For every  $K \in \mathcal{K}$ ,  $W \in \mathcal{W}$ , and  $X \in \{0, 1\}^n \setminus \mathcal{W}$

$$\begin{aligned} E_K(W) &\in \mathcal{W}, \\ E_K(X) &\in \{0, 1\}^n \setminus \mathcal{W}. \end{aligned}$$

Similar to the ideal cipher model, if a security proof supposes a weak block cipher is picked uniformly random from  $\Pi_w(k, n, \mathcal{K}, \mathcal{W})$  at the beginning of the query, we call the proof is under *weakly ideal cipher model*.

DRBG. From [9, 18, 7], a DRBG (Deterministic Random Bit Generator) is a triple of algorithms  $\mathcal{G} = (\text{setup}, \text{refresh}, \text{next})$  where:

- **setup** : an algorithm that outputs an initial state  $S$ .
- **refresh** : an entropy accumulating algorithm that, given a state  $S$  and an input  $I$ , outputs a new state  $S'$ .
- **next** : a random bit generating algorithm that, given a state  $S$ , outputs a new state  $S'$  and random bits  $R$ .

However, the Linux-DRBG do not fully fit in the above DRBG model, it has two **refresh** functions and two **next** functions. These functions are described in Algorithm 4 and we explained the reason that we modeled Linux-DRBG in this format in Section 4.

**DISTINGUISH GAME.** Throughout this paper, we prove the robustness of Linux-DRBG by showing that Linux-DRBG and its subalgorithms are **indistinguishable** from an ideal DRBG with a distinguishing game. Generally, let  $\mathcal{G}_0$  and  $\mathcal{G}_1$  be algorithms and  $\mathcal{A}$  be an adversary to distinguish them. Then the distinguish game is composed as below.

1.  $b \leftarrow_{\$} \{0, 1\}$ , then  $\mathcal{G}_b$  make interfaces that  $\mathcal{A}$  can access and get response. The interfaces are called oracles.
2. Under the prescribed rule,  $\mathcal{A}$  makes queries to the oracle and gets responses.
3. After querying phase,  $\mathcal{A}$  outputs  $b'$ . If  $b' = b$ ,  $\mathcal{A}$  wins.

Let the distinguishing game between  $\mathcal{G}_0$  and  $\mathcal{G}_1$  be **dist**. Then the distinguishing advantage of  $\mathcal{A}$  against **dist**,  $\text{Adv}_{\text{dist}}(\mathcal{A})$  is defined as below.

$$\begin{aligned} \text{Adv}_{\text{dist}}(\mathcal{A}) &= |\Pr[1 \leftarrow \mathcal{A}|b = 0] - \Pr[1 \leftarrow \mathcal{A}|b = 1]| \\ &= |\Pr[0 \leftarrow \mathcal{A}|b = 0] - \Pr[0 \leftarrow \mathcal{A}|b = 1]|. \end{aligned}$$

**H-COEFFICIENT TECHNIQUE.** At the end of the distinguishing game, an adversary obtains a certain transcript, containing all the information obtained during the attack. A transcript is called *attainable* if the probability of obtaining it in the ideal world is non-zero.

**Lemma 1 (Patarin’s H-coefficient technique).** *Let  $\mathcal{T}$  be the set of all attainable transcripts. Suppose that  $\mathcal{T}$  is partitioned as  $\mathcal{T} = \mathcal{T}_{\text{good}} \sqcup \mathcal{T}_{\text{bad}}$  for two subsets  $\mathcal{T}_{\text{good}}$  and  $\mathcal{T}_{\text{bad}}$ , where*

$$\text{pid}[\tau \in \mathcal{T}_{\text{bad}}] \leq \epsilon_1$$

*for some  $\epsilon_1 \geq 0$ , and there exists  $\epsilon_2 \geq 0$  such that*

$$\frac{\text{pre}[\tau]}{\text{pid}[\tau]} \geq 1 - \epsilon_2$$

for any  $\tau \in \mathcal{T}_{\text{good}}$ . Then for any distinguisher  $\mathcal{A}$ , one has

$$\mathbf{Adv}_{\text{dist}}(\mathcal{A}) \leq \epsilon_1 + \epsilon_2.$$

MIN ENTROPY. Let the prediction probability of a random variable  $X$  be

$$\text{Pred}(X) := \max_x \Pr[X = x].$$

Then for another random variable  $Y$ ,  $\text{Pred}(X|y) := \max_x \Pr[X = x|Y = y]$ . Then conditional probability of  $X$  over  $Y$  is

$$\text{Pred}(X|Y) := E(\text{Pred}(X|y)).$$

And (average-case) conditional min-entropy is

$$H_{\infty}(X|Y) = -\log(\text{Pred}(X|Y)).$$

### 3 Seedless Robustness Model

The seedless robustness model [7] is a modification of the “seeded” robustness model [9], designed to relax the unrealistic assumption of the original model, namely, the existence of a random seed that should be kept secret to an adversary and independent of the entropy source.

SEEDLESS ROBUSTNESS ORACLE. Let

$$\mathcal{G} = (\text{setup}[P], \text{refresh}[P], \text{next}[P])$$

be a DRBG based on an ideal primitive  $P$  (such as a random oracle, an ideal cipher, or a random permutation), where  $P$  is chosen uniformly at random from the set of all possible primitives, denoted  $\mathcal{P}$ . Then the seedless robustness oracles are defined as described in Algorithm 1, where  $c$  denotes the entropy accumulated in the DRBG and  $\lambda$  is a fixed parameter (denoting the minimum required for the accumulated entropy).

SEEDLESS ADVERSARY. In seedless robustness model [7], an adversary  $\mathcal{A}$  consists of two algorithms  $\mathcal{A}_1$  and  $\mathcal{A}_2$ . The relationship between  $\mathcal{A}_1$  and  $\mathcal{A}_2$  is as follows:

- $\mathcal{A}_1$  is allowed to make queries only to the entropy accumulating oracle REF, while  $\mathcal{A}_2$  is allowed to make queries to all the other oracles except REF,
- $\mathcal{A}_1$  knows all the queries made by  $\mathcal{A}_2$  and the corresponding responses, while  $\mathcal{A}_2$  observes only the responses to the queries made by  $\mathcal{A}_1$  without knowing the queries themselves.

$\mathcal{A}_1$  is modeled in a way that the adversary can influence entropy accumulation but cannot ascertain specific values of entropy inputs. In the “seeded” robustness model [9], the distribution sampler  $\mathcal{D}$  provides entropy inputs. However, as argued in [7], the security proof using  $\mathcal{D}$  is based on an unrealistic

**Algorithm 1** Oracles for Seedless Robustness Game

---

<b>Procedure</b> INIT() 1: $b \leftarrow_{\$} \{0, 1\}$ 2: $P \leftarrow_{\$} \mathcal{P}, c \leftarrow 0$ 3: $S \leftarrow \text{setup}[P]()$ 4: <b>return</b> $P$	
<b>Procedure</b> REF( $I, \gamma$ ) 1: $S \leftarrow \text{refresh}[P](S, I); c \leftarrow c + \gamma$ 2: <b>return</b> $\gamma$	<b>Procedure</b> ROR( $len$ ) 1: $y_0 \leftarrow_{\$} \{0, 1\}^{len}$ 2: $(S, y_1) \leftarrow \text{next}[P](S, len)$ 3: <b>if</b> $c < \lambda$ <b>then</b> 4: $c \leftarrow 0$ ; <b>return</b> $y_1$ 5: <b>return</b> $y_b$
<b>Procedure</b> GET(); 1: $c \leftarrow 0$ ; <b>return</b> $S$	<b>Procedure</b> SET( $S^*$ ) 1: $S \leftarrow S^*; c \leftarrow 0$
<b>Procedure</b> $P(x)$ 1: <b>return</b> $P(x)$	<b>Procedure</b> $P^{-1}(x)$ //If exists 1: <b>return</b> $P^{-1}(x)$

---

assumption that  $\mathcal{D}$  is independent of the underlying primitive of the DRBG (In Linux-DRBG,  $E$ , and  $\pi$ ). Instead, they replaced the distribution sampler with  $\mathcal{A}_1$  that only accumulates entropy in the DRBG.

To model the quality of the entropy source, we define legitimacy for  $\mathcal{A}$ . Let  $\mathcal{I}_i$  be the random variable for  $i$ -th input  $\mathcal{A}_1$  makes, and  $\mathcal{T}_i$  be the random variable for all input-output list of robustness game, excluding  $I_i$ , the  $i$ -th entropy input. Then  $\mathcal{A}$  is  $\gamma^*$ -**legitimate** if

$$H_\infty(\mathcal{I}_i | \mathcal{T}_i) \geq \gamma_i \geq \gamma^*.$$

for every  $i$ . Against a  $\gamma^*$ -*legitimate* adversary  $\mathcal{A}$ , the seedless robust game is defined as follows.

**Seedless Robustness Game.**

1. INIT is executed.
2.  $\mathcal{A}_1$  makes queries to REF, while  $\mathcal{A}_2$  makes queries to ROR, GET, SET,  $P$  and  $P^{-1}$  (if available) in an interleaved manner.
3.  $\mathcal{A}_2$  outputs  $b' \in \{0, 1\}$ .
4. If  $b' = b$ , then  $\mathcal{A}$  wins, and  $\mathcal{A}$  loses otherwise.

The game begins with the procedure INIT, which chooses a random bit  $b$  and a random primitive  $P$ . It then runs **setup** to make the initial state. The other parts of the game are oracles offered to an adversary  $\mathcal{A}$ :

- REF : state  $S$  is updated by calling `refresh` with input  $I$  of entropy at least  $\gamma$ ,
- ROR : state  $S$  is updated by calling `next`, and then  $y_0$  and  $y_1$  are prepared: if  $c < \lambda$ , then it returns  $y_1$  regardless of  $b$ , while if  $c \geq \lambda$ , then  $y_b$  is returned
- GET : returns the state  $S$  of the DRBG.
- SET( $S^*$ ) : sets the state  $S$  of the DRBG to  $S^*$ .
- $P$  returns the result of a primitive query. If the primitive allows inverse query (i.e., block cipher), the oracle should also provide  $P^{-1}$ .

Now we can distinguish two worlds  $\mathcal{G}_0$  and  $\mathcal{G}_1$  from the seedless robust games according to the bit  $b \in \{0, 1\}$ , and for any DRBG  $\mathcal{G}$ , the seedless robustness security of  $\mathcal{G}$  against  $\mathcal{A}$  is defined as follows.

$$\mathbf{Adv}_{rob}^{\mathcal{G}}(\mathcal{A}) \stackrel{\text{def}}{=} \mathbf{Adv}_{\text{dist}}(\mathcal{A}).$$

## 4 Modeling the Linux DRBG

In this section, we model the Linux DRBG to fit into the Seedless Robustness security model. In this paper, we studied the Linux version 6.4.8.<sup>4</sup> We mainly analyzed a `random.c` file in Linux 6.4.8, a collection of functions related to the Linux DRBG. The Linux DRBG uses the hash function BLAKE2s and the stream cipher ChaCha20 as internal primitives. We first describe the modeling of the two internal primitives and then explain the overall structure of the Linux DRBG.

### 4.1 Underlying primitives and their modeling

---

#### Algorithm 2 A Procedure in the BLAKE2s

---

COMP :  $\{0, 1\}^{n/4} \times \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n$   
**Procedure** COMP[ $E$ ]( $t, h, I$ )  
1:  $len \leftarrow |I|$   
2:  $rem \leftarrow len - 2n(\lceil len/2n \rceil - 1)$   
3:  $(I_1, \dots, I_\ell) \xleftarrow{2n} I$   
4:  $I_\ell \leftarrow 0^{2n-rem} \parallel I_\ell$   
5:  $h_1 \leftarrow h$   
6: **for**  $i \leftarrow 1$  to  $\ell - 1$  **do**  
7:      $h_{i+1} \leftarrow B[E](h_i, t + i \cdot 2n, I_i)$   
8:  $y \leftarrow B'[E](h_\ell, t + len, I_\ell)$   
9: **return**  $y$

---

<sup>4</sup> <https://github.com/torvalds/linux/blob/master/drivers/char/random.c> 



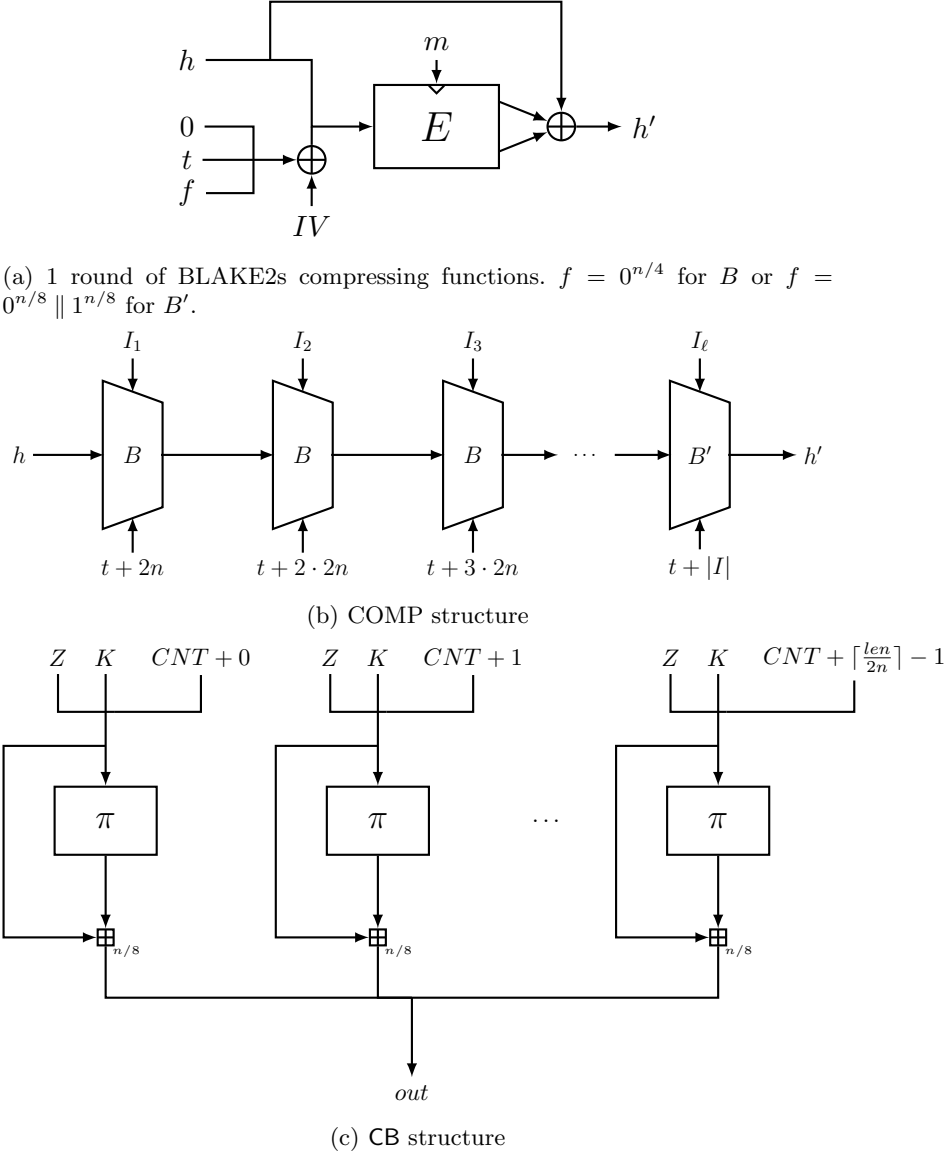


Fig. 2: BLAKE2s and ChaCha20 internal structure

**4.1.1 BLAKE2s** The Linux DRBG accumulates entropy using the hash function BLAKE2s [1].

**THE WEAKLY IDEAL CIPHER MODEL.** Due to the known structure of the compression function  $E$  of the BLAKE2s [15], we cannot model  $E$  as an ideal cipher. Hence, we model  $E$  as a weak block cipher following the [15]. For the BLAKE2s,

a weak key set  $\mathcal{K}$  and a weak input set  $\mathcal{W}$  are defined as follows:

$$\begin{aligned}\mathcal{W} &= \{aaaabbbbccccdddd \in \{0, 1\}^{2n} \mid a, b, c, d \in \{0, 1\}^{n/8}\}, \\ \mathcal{K} &= \{kkkkkkkkkkkkkkkk \in \{0, 1\}^{2n} \mid k \in \{0, 1\}^{n/8}\}.\end{aligned}$$

Note that  $|\mathcal{W}| = 2^{0.5n}$ . Then, a set of weak block ciphers in the BLAKE2s is  $\Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$ .

**THE MODELING OF THE COMPRESSION FUNCTION.** For  $x \in \{0, 1\}^{2n}$ , let  $\text{TRSum}(x) = x[0 : n-1] \oplus x[n : ]$ . Then the 1 round of compression function of the BLAKE2s is defined as follows:

$$\begin{aligned}B[E](h, t, I) &\leftarrow \text{TRSum}(E(I, h \parallel (0^{n/2} \parallel t \parallel 0^{n/4}) \oplus IV)) \oplus h, \\ B'[E](h, t, I) &\leftarrow \text{TRSum}(E(I, h \parallel (0^{n/2} \parallel t \parallel 0^{n/8} \parallel 1^{n/8}) \oplus IV)) \oplus h\end{aligned}$$

where  $E \in \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$ ,  $h$  is a  $n$ -bit value,  $t$  is a  $n/4$ -bit counter,  $I$  is an  $2n$ -bit input, and  $IV = IV_1 \parallel \dots \parallel IV_8$  is  $n$ -bit fixed string where  $IV_k \in \{0, 1\}^{n/8}$  for all  $1 \leq k \leq 8$  and  $IV_i \neq IV_j$  for all  $1 \leq i \neq j \leq 8$ .  $B[E](h, t, I)$  and  $B'[E](h, t, I)$  are described in Figure 2.(a). Here,  $B$  is used when  $I$  is not the final input block, and  $B'$  is used when  $I$  is the final input block. We represent the Linux DRBG function `blake2s_compress` as COMP in Algorithm 2 and Figure 2.(b).

**AVOIDING THE WEAK STATE.** In BLAKE2s, the  $IV$  is a 256-bit string consisting of a tuple of 32-bit values derived from the square roots of distinct primes starting from 2 and ending at 19. Note that the square roots of primes are all different. Hence, in actual usage, the weak input  $w \in \mathcal{W}$  of BLAKE2s cannot be accessed due to the distinct elements of  $IV$ . Therefore, we assume that the  $IV = IV_1 \parallel \dots \parallel IV_8$  is  $n$ -bit fixed string where  $IV_k \in \{0, 1\}^{n/8}$  for all  $1 \leq k \leq 8$  and  $IV_i \neq IV_j$  for all  $1 \leq i \neq j \leq 8$ .

---

**Algorithm 3** A Procedure in the ChaCha20

---

$\text{CB} : \{0, 1\}^n \times \{0, 1\}^{n/2} \times \{0, 1\}^* \rightarrow \{0, 1\}^*$   
**Procedure**  $\text{CB}[\pi](K, CNT, len)$   
1:  $out \leftarrow \epsilon$   
2: **while**  $len > 0$  **do**  
3:    $B \leftarrow \pi(Z \parallel K \parallel CNT) +_{n/8} (Z \parallel K \parallel CNT)$   
4:    $CNT \leftarrow CNT + 1$   
5:    $out \leftarrow out \parallel B$   
6:    $len \leftarrow len - 2n$   
7: **return**  $out$

---

**4.1.2 ChaCha20** The Linux DRBG produces pseudorandom outputs using ChaCha20. The stream cipher ChaCha20 is known to be faster than AES when

hardware support for AES is not available. The ChaCha20 uses a fixed constant  $Z$ , expressed as the hexadecimal representation of “expand 32-byte k”.

**THE RANDOM PERMUTATION MODEL.** The 20 rounds of the ChaCha20 can be modeled as a random permutation  $\pi \leftarrow_{\$} \text{Perm}(2n)$  [8]. Using the random permutation  $\pi$  and a fixed constant  $Z$ , we model a function `chacha_block` as `CB` in the Algorithm 3 and Figure 2.(c). The computation of `CB` on one  $2n$  bits block is expressed as follows:

$$out \leftarrow \pi(Z \parallel K \parallel CNT) +_{n/8} (Z \parallel K \parallel CNT)$$

where,  $Z$  is the fixed  $n/2$ -bit constant,  $K$  is an  $n$ -bit key, and  $CNT$  is an  $n/2$ -bit counter, and  $+_{n/8}$  represents word-by-word modulo  $2^{n/8}$  addition.

## 4.2 The Overall Structure of the Linux DRBG

We divide the Linux DRBG into three parts: Initialization, Entropy Accumulation, and Random Bit Generation. The Linux DRBG is initialized when the Linux kernel boots. Then, the Linux DRBG starts to accumulate entropy from various hardware sources. When accumulated entropy is larger than 256-bit, a character device file `/dev/random`, a system call `getrandom`, and a kernel interface `get_random_bytes` of the Linux DRBG can produce random bits. In this subsection, we describe the three parts of the Linux DRBG. We also describe how we model each part in the Seedless Robustness model which will be described in the Algorithm 4 and Figure 3 later.

**4.2.1 Initialization** When the Linux kernel boots, the Linux DRBG initializes a state `input_pool` of the BLAKE2s. Also, the Linux DRBG initializes states `base_crng` and `crng` of the ChaCha20. Then the Linux DRBG calls a function `random_init_early` that accumulates entropy in the `input_pool` without accessing the timer in Linux. Finally, when the timer becomes available, the Linux DRBG calls a function `random_init` which accumulates entropy in the `input_pool` using the timer in Linux.

**THE MODELING OF THE INITIALIZATION.** We model the three states `input_pool`, `base_crng`, and `crng` in a single state  $S$ . Using the state  $S$ , we model the initialization as a function `setup()` in the Algorithm 4. Note that the Linux DRBG accumulates some entropy in the initialization. But we exclude entropy accumulation in the `setup`. Then, we model an attacker to call entropy accumulation functions with high entropy after entropy-draining events including `setup`.

**4.2.2 Entropy Accumulation** The Linux DRBG accumulates entropy from various hardware entropy sources. The Linux DRBG calls functions to access hardware entropy sources. The functions related to entropy accumulation and estimation are listed as follows:

- `add_hwgenerator_randomness`,

- `add_bootloader_randomness`,
- `add_interrupt_randomness`,
- `add_timer_randomness`.

When the Linux DRBG calls the functions, they return a string that contains entropy which is called “entropy input”. Note that these functions also return an estimation of the entropy within their entropy inputs. The Linux DRBG credits the estimation using a function `credit_init_bits`, enabling it to track how much entropy has been accumulated in the state of the DRBG. An analysis of Linux kernel version 5.18.1 by the German Federal Office for Information Security (BSI) shows that the Linux entropy sources and their entropy estimations satisfy their security criteria [16]. Hence, we assume that all entropy sources and estimations are functioning correctly.

**ENTROPY ACCUMULATING FUNCTIONS.** The Linux DRBG utilizes the function `mix_pool_bytes` to use the BLAKE2s for accumulating entropy into its state. Also, the Linux DRBG uses a function `crng_reseed` to convert the BLAKE2s’ state into a key for a random bit generation. Entropy accumulation works as follows:

1. The Linux DRBG obtains an entropy input from the entropy sources.
2. The Linux DRBG passes the entropy input to the hash function BLAKE2s.
3. `mix_pool_bytes`: BLAKE2s compresses the entropy input to its element  $h$  of its state.
4. `crng_reseed`: If the Linux DRBG needs to generate random bits, then the BLAKE2s uses  $h$  to derive keys for the random bit generation.

**THE MODELING OF THE ENTROPY ACCUMULATION.** We put constants  $h$ ,  $t$ , and key  $k_{\text{next}}$  of the BLAKE2s in the DRBG state  $S$ . Also, we put ChaCha20 key  $k_{\text{base}}$  in the state  $S$ . In the Algorithm 4, the two entropy accumulating functions `mix_pool_bytes` and `crng_reseed` are modeled as `refresha` and `refreshf`. We depict the two functions in the Figure 3.(a) and (b). In the Figure 3.(a), the `refresha` uses an entropy input  $I$  to update  $h$ . In the Figure 3.(b), the `refreshf` uses  $h$  and an entropy input  $I$  to generate  $k_{\text{next}}$  for later BLAKE2s calls and  $k_{\text{base}}$  for random bit generation.

**4.2.3 Random Bit Generation** The Linux DRBG uses either a function `get_random_bytes` or a function `get_random_bytes_user` to utilize the ChaCha20 for generating random bits. Note that Linux uses two types of the ChaCha20 for a multi-core system. A `base_crng` obtains the key  $k_{\text{base}}$  from the BLAKE2s and produces `ckeyCPU`. A `crng`, within each CPU, generates random bits utilizing the CPU with a key `ckeyCPU`. Random Bit Generation works as follows:

1. The Linux DRBG obtains the ChaCha20 key  $k_{\text{base}}$  from the BLAKE2s and initializes the `base_crng` using the  $k_{\text{base}}$ .
2. The Linux DRBG assigns a CPU the random bit generation task.

3. (a) If a flag  $G\_flag_{CPU}$  is set, Linux generates the CPU-specific key  $ckey_{CPU}$  from the  $base\_crng$ .  
 (b) Otherwise, Linux updates  $ckey_{CPU}$  by running  $crng$  with the old  $ckey_{CPU}$  without accessing  $k_{base}$ .
4. Using the  $crng$  in the CPU, the Linux DRBG uses either the `get_random_bytes` or the `get_random_bytes_user` to generate random bits.

TWO TYPES OF RANDOM BIT GENERATORS. In the Linux DRBG, there are two types of random bit generation:

1. A first type generator produces random bits only if sufficient entropy is accumulated. The `/dev/random` is a representative example of the type.
2. A second type generator produces random bits at any time. The `/dev/urandom` is a representative example of the type.

The only difference between `/dev/random` and `/dev/urandom` is the fact that `/dev/random` prohibits *premature next* and `/dev/urandom` allows it. There already exists the study that any DRBG that allows *premature next* is insecure in the seedless model [6]. Therefore in this paper, we only consider the first type. The character device file `/dev/random`, the system call `getrandom`, and the kernel interface `get_random_bytes` are the first type generators. The `/dev/random` and the `getrandom` use the function `get_random_bytes_user` to generate random bits. The `get_random_bytes` kernel interface uses the function `get_random_bytes` to generate random bits. These generators can produce random bits only if accumulated entropy is more than 256-bit.

THE MODELING OF THE RANDOM BIT GENERATION. We put keys  $k_{base}$ ,  $ckey_1$ ,  $\dots$ ,  $ckey_C$ , and flags  $G\_flag_1$ ,  $\dots$ ,  $G\_flag_C$  in the state  $S$  where  $C$  is the number of the CPUs. In the Algorithm 4, two random bit generation functions `get_random_bytes` and `get_random_bytes_user` are modeled as  $next_k$  and  $next_u$ . We depict the two functions in the Figure 3.(c) and (d).

### 4.3 Syntax of Linux DRBG in Robustness Model

From the previous subsection, we establish the state of the Linux DRBG. Building upon this state, we present the syntax of Linux DRBG, which constitutes our model of the operation of Linux DRBG.

**4.3.1 Internal State** We define the internal state of the Linux DRBG  $S$  as follows:

$$S := (h, t, k_{next}, k_{base}, ckey_1, \dots, ckey_C, G\_flag_1, \dots, G\_flag_C)$$

where  $C$  is the number of available CPUs. A description of each element in the state  $S$  is as follows:

- $h \in \{0, 1\}^n$ : a value that is updated by the compression function of the BLAKE2s,

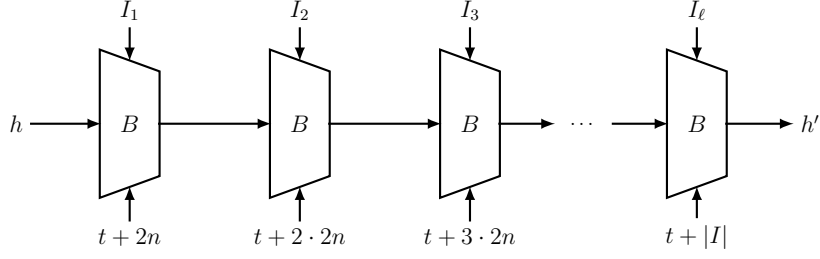
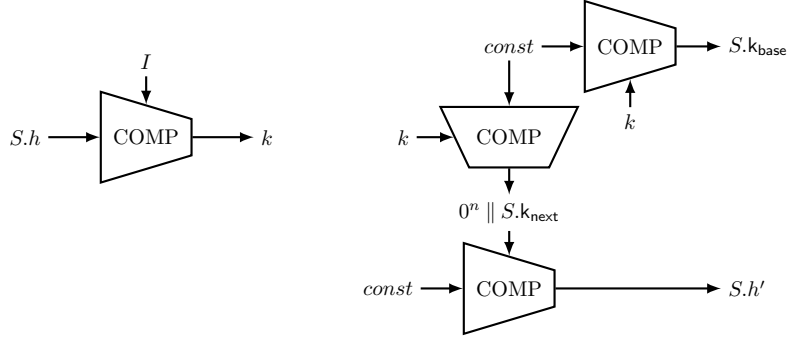
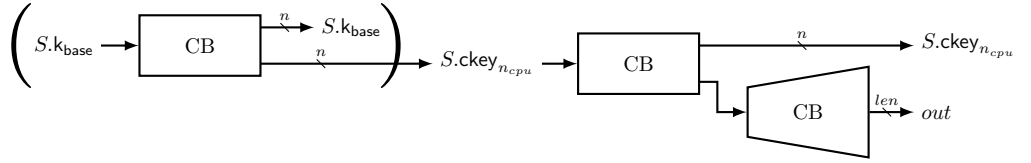
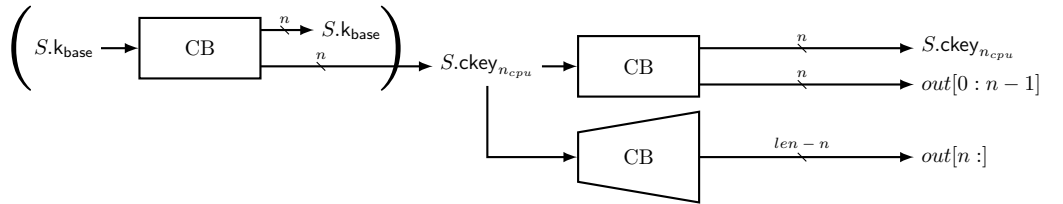

 (a)  $\text{refresh}_a$ 

 (b)  $\text{refresh}_f$ 

 (c)  $\text{next}_u$ 

 (d)  $\text{next}_k$ 

 Fig. 3: Components of Linux-DRBG. States are defined in Section 4.3 and  $B$ ,  $\text{CB}$  and  $\text{COMP}$  are defined in Section 4.1

**Algorithm 4** Syntax of the Linux DRBG

---

```

    setup :  $\epsilon \rightarrow \mathcal{S}$ 
    Procedure setup()
1:  $S.h \leftarrow \text{const}; S.t \leftarrow 0$ 
2: return  $S$ 

    refresha :  $\mathcal{S} \times (\{0, 1\}^{2n})^* \rightarrow \mathcal{S}$ 
    Procedure refresha[ $E$ ]( $S, I$ )
1: if  $S.k_{\text{next}} \neq \epsilon$  then
2:    $I \leftarrow 0^n \parallel S.k_{\text{next}} \parallel I$ 
3:    $S.k_{\text{next}} \leftarrow \epsilon$ 
4:    $(I_1, \dots, I_\ell) \xleftarrow{2n} I$ 
5:   for  $i \leftarrow 1$  to  $\ell$  do
6:      $S.h \leftarrow B(S.h, S.t + i \cdot 2n, I_i)$ 
7:    $S.t \leftarrow S.t + |\ell|$ 
8:   return  $S$ 

    refreshf :  $\mathcal{S} \times \{0, 1\}^* \times \{0, 1\}^n \rightarrow \mathcal{S}$ 
    Procedure refreshf[ $E$ ]( $S, I, I_{\text{cpu}}$ )
1: if  $S.k_{\text{next}} \neq \epsilon$  then
2:    $I \leftarrow 0^n \parallel S.k_{\text{next}} \parallel I$ 
3:    $k \leftarrow \text{COMP}[E](S.t, S.h, I)$ 
4:    $I_{\text{cpu}} \leftarrow 0^n \parallel k \parallel I_{\text{cpu}} \parallel 0^{c-1}$ 
5:    $S.k_{\text{next}} \leftarrow \text{COMP}[E](0, \text{const}, I_{\text{cpu}} \parallel 0)$ 
6:    $S.k_{\text{base}} \leftarrow \text{COMP}[E](0, \text{const}, I_{\text{cpu}} \parallel 1)$ 
7:    $S.h \leftarrow \text{const}; S.t \leftarrow 2n$ 
8:    $S.G.\text{flag}_1 \leftarrow 1; S.G.\text{flag}_2 \leftarrow 1; \dots;$ 
      $S.G.\text{flag}_C \leftarrow 1$ 
9:   return  $S$ 

    nextk :  $\mathcal{S} \times \{0, 1\}^* \times \{0, 1\}^{\log(C)} \rightarrow \mathcal{S} \times \{0, 1\}^*$ 
    Procedure nextk[ $\pi$ ]( $S, \text{len}, n_{\text{cpu}}$ )
1: if  $S.G.\text{flag}_{n_{\text{cpu}}} = 1$  then
2:    $\text{tmp} \leftarrow \text{CB}[\pi](S.k_{\text{base}}, 0, 2n)$ 
3:    $S.k_{\text{base}} \parallel S.\text{ckey}_{n_{\text{cpu}}} \leftarrow \text{tmp}$ 
4:    $S.G.\text{flag}_{n_{\text{cpu}}} \leftarrow 0$ 
5:    $k \leftarrow S.\text{ckey}_{n_{\text{cpu}}}$ 
6:    $\text{tmp} \leftarrow \text{CB}[\pi](S.\text{ckey}_{n_{\text{cpu}}}, 0, 2n)$ 
7:    $S.\text{ckey}_{n_{\text{cpu}}} \parallel \text{out} \leftarrow \text{tmp}$ 
8:    $B \leftarrow \text{CB}[\pi](k, 1, \text{len} - n)$ 
9:    $\text{out} \leftarrow (\text{out} \parallel B)[0 : \text{len} - 1]$ 
10: return  $(S, \text{out})$ 

    nextu :  $\mathcal{S} \times \{0, 1\}^* \times \{0, 1\}^{\log(C)} \rightarrow \mathcal{S} \times \{0, 1\}^*$ 
    Procedure nextu[ $\pi$ ]( $S, \text{len}, n_{\text{cpu}}$ )
1: if  $S.G.\text{flag}_{n_{\text{cpu}}} = 1$  then
2:    $\text{tmp} \leftarrow \text{CB}[\pi](S.k_{\text{base}}, 0, 2n)$ 
3:    $S.k_{\text{base}} \parallel S.\text{ckey}_{n_{\text{cpu}}} \leftarrow \text{tmp}$ 
4:    $S.G.\text{flag}_{n_{\text{cpu}}} \leftarrow 0$ 
5:    $\text{tmp} \leftarrow \text{CB}[\pi](S.\text{ckey}_{n_{\text{cpu}}}, 0, 2n)$ 
6:    $S.\text{ckey}_{n_{\text{cpu}}} \parallel k \leftarrow \text{tmp}$ 
7:   if  $\text{len} \leq n$  then
8:     return  $(S, k[0 : \text{len} - 1])$ 
9:    $\text{out} \leftarrow (\text{CB}[\pi](k, 1, \text{len}))[0 : \text{len} - 1]$ 
10: return  $(S, \text{out})$ 

```

---

- $t \in \{0, 1\}^{n/4}$ : a counter of the BLAKE2s,
- $k_{\text{next}} \in \{0, 1\}^n$ : a key of the BLAKE2s,
- $k_{\text{base}} \in \{0, 1\}^n$ : a key of the ChaCha20 that is used to produce CPU-specific keys  $\text{ckey}_1, \dots, \text{ckey}_C$ ,
- $\text{ckey}_i \in \{0, 1\}^n$ : a key of the ChaCha20 that is used to produce random bits in the  $i$ -th CPU,
- $G.\text{flag}_i \in \{0, 1\}$ : a flag that indicates whether  $\text{ckey}_i$  needs to be updated by using  $k_{\text{base}}$  or not.

We define a set of states as

$$\mathcal{S} = \{0, 1\}^n \times \{0, 1\}^{n/4} \times \{0, 1\}^n \times \{0, 1\}^n \times (\{0, 1\}^n)^C \times (\{0, 1\})^C.$$

**MODELING THE BLAKE2S STATE.** The compression function of BLAKE2s updates  $S.h \in \{0, 1\}^n$ , and  $S.t \in \{0, 1\}^{n/4}$  serves as an input to BLAKE2s, accumulating the bit length of the input compressed. In Linux, BLAKE2s maintains a buffer and finalization flag in its state. The value to be compressed is stored in the buffer, and when Linux DRBG reseeds, it initializes BLAKE2s' state and places a key in the buffer. We eliminated the buffer from  $S$  and stored the BLAKE2s key in  $S.k_{\text{next}}$ . By saving  $S.k_{\text{next}}$ , we can simulate BLAKE2s without the buffer. For the finalization flag in Linux, we explicitly incorporate it into the COMP in the Algorithm 2.

**MODELING THE CHACHA20 STATE.** The final output of the BLAKE2s serves as the key for the ChaCha20, denoted as  $S.k_{\text{base}} \in \{0, 1\}^n$ . To leverage a multi-core system, each CPU has its ChaCha20 states. Their keys are stored in  $\text{ckey}_1, \dots, \text{ckey}_C \in \{0, 1\}^n$ , where  $C$  represents the maximum available CPU number. When Linux DRBG is asked to produce pseudorandom outputs, it determines whether the  $i$ -th CPU ChaCha20's key needs to be updated by using  $S.k_{\text{base}}$  or not based on  $G.\text{flag}_1, \dots, G.\text{flag}_C \in \{0, 1\}$ . If  $G.\text{flag}_i = 1$ , then it needs to be updated using  $S.k_{\text{base}}$ . Otherwise, it is updated using old  $S.\text{ckey}_i$ .

**4.3.2 Linux DRBG Syntax** We align the Linux DRBG with the syntax of the PRNG used in the robustness model [9,7]. The syntax of the Linux DRBG is detailed in Algorithm 4. Additionally, we illustrate the operations of each function in Figure 3.

- **setup()**: This algorithm produces an initial Linux DRBG state  $S$ .
- **refresh<sub>a</sub>[ $E$ ]( $S, I$ )**: Given a weakly ideal cipher  $E$ , the state  $S$  and a variable length entropy input  $I$ , **refresh<sub>a</sub>** compresses the entropy input  $I$  and store it to  $S.h$ .
- **refresh<sub>f</sub>[ $E$ ]( $S, I, I_{\text{cpu}}$ )**: Given a weakly ideal cipher  $E$ , the state  $S$ , a variable length entropy input  $I$ , and a fixed  $n$ -bit input  $I_{\text{cpu}}$ , **refresh<sub>f</sub>** compresses  $I$  and  $I_{\text{cpu}}$  to generate two keys  $S.k_{\text{next}}$  and  $S.k_{\text{base}}$ . Note that **refresh<sub>f</sub>** uses fixed constant  $\text{const}$  for compression. The  $\text{const}$  is defined as follows:

$$\text{const} \leftarrow IV;$$

$$\text{const}[0 : n/8 - 1] \leftarrow \text{const}[0 : n/8 - 1] \oplus (0^7 \parallel 1 \parallel 0^7 \parallel 1 \parallel 0^{n/8-16} | n < 8 | n).$$

where the  $IV$  is the fixed constant from the BLAKE2s, and  $|$  is the bit-wise OR operation.  $S.k_{\text{next}}$  is used in later calls to **refresh<sub>a</sub>** or **refresh<sub>f</sub>**.  $S.k_{\text{base}}$  is used to generate random bits through **next<sub>k</sub>** or **next<sub>u</sub>**. The **refresh<sub>f</sub>** also sets flags  $S.G.\text{flag}_1, \dots, S.G.\text{flag}_C$ .

- **next<sub>k</sub>[ $\pi$ ]( $S, \text{len}, n_{\text{cpu}}$ )**: Given a random permutation  $\pi$ , the state  $S$ , a required output length  $\text{len}$ , and a CPU number  $n_{\text{cpu}}$ , **next<sub>k</sub>** generates  $\text{len}$ -bit random



- bits using  $S.\text{ckey}_{n_{cpu}}$  in the  $n_{cpu}$ -th CPU. If a  $S.G.\text{flag}_{n_{cpu}}$  is set, then  $\text{next}_k$  updates  $S.\text{ckey}_{n_{cpu}}$  by using  $S.k_{\text{base}}$ . Otherwise,  $\text{next}_k$  updates  $S.\text{ckey}_{n_{cpu}}$  by its old  $S.\text{ckey}_{n_{cpu}}$ .
- $\text{next}_u[\pi](S, \text{len}, n_{cpu})$ : This algorithm works similar to the  $\text{next}_k$ . The  $\text{next}_k$  directly uses  $S.\text{ckey}_{n_{cpu}}$  to produce random bits. But the  $\text{next}_u$  first produces a temporary key  $k$ , then it produces random bits using the  $k$ .

## 5 Robustness Proof

### 5.1 Linux-DRBG Robustness Game

---

**Algorithm 5** Oracles for Linux-DRBG Seedless Robustness Game

---

**Procedure** INIT()

- 1:  $b \leftarrow_{\$} \{0, 1\}, c \leftarrow 0$
- 2:  $E \leftarrow_{\$} \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W}), \pi \leftarrow_{\$} \text{Perm}(2n)$
- 3:  $S \leftarrow \text{setup}()$
- 4: **return**  $(E, \pi)$

**Procedure** REF<sub>a</sub>[ $E$ ]( $I, \gamma$ )

- 1:  $S \leftarrow \text{refresh}_a[E](S, I); c \leftarrow c + \gamma$
- 2: **return**  $\gamma$

**Procedure** ROR<sub>k</sub>[ $\pi$ ]( $\text{len}, n_{cpu}$ )

- 1:  $(S, y_1) \leftarrow \text{next}_k[\pi](S, \text{len}, n_{cpu})$
- 2: **if**  $c < \lambda$  or  $\text{ready} = 0$  **then**
- 3:    $c \leftarrow 0; \text{ready} \leftarrow 0; \text{return } y_1$
- 4:  $y_0 \leftarrow_{\$} \{0, 1\}^{\text{len}}$
- 5: **return**  $y_b$

**Procedure** GET();

- 1:  $c \leftarrow 0; \text{ready} \leftarrow 0; \text{return } S$

**Procedure**  $E(k, x)$ 

- 1: **return**  $E(k, x)$

**Procedure**  $\pi(x)$ 

- 1: **return**  $\pi(x)$

**Procedure** REF<sub>f</sub>[ $E$ ]( $I, \gamma, I_{cpu}$ )

- 1:  $S \leftarrow \text{refresh}_f[E](S, I, I_{cpu})$
- 2:  $c \leftarrow c + \gamma$
- 3: **if**  $c \geq \lambda$  **then**
- 4:    $\text{ready} \leftarrow 1$
- 5: **else**
- 6:    $c \leftarrow 0$
- 7: **return**  $\gamma$

**Procedure** ROR<sub>u</sub>[ $\pi$ ]( $\text{len}, n_{cpu}$ )

- 1:  $(S, y_1) \leftarrow \text{next}_u[\pi](S, \text{len}, n_{cpu})$
- 2: **if**  $c < \lambda$  or  $\text{ready} = 0$  **then**
- 3:    $c \leftarrow 0; \text{ready} \leftarrow 0; \text{return } y_1$
- 4:  $y_0 \leftarrow_{\$} \{0, 1\}^{\text{len}}$
- 5: **return**  $y_b$

**Procedure** SET( $S^*$ )

- 1:  $S \leftarrow S^*; c \leftarrow 0; \text{ready} \leftarrow 0$

**Procedure**  $E^{-1}(k, y)$ 

- 1: **return**  $E^{-1}(k, y)$

**Procedure**  $\pi^{-1}(y)$ 

- 1: **return**  $\pi^{-1}(y)$
-

**DIFFERENCE FROM THE GENERAL DRBG MODEL.** From procedures in Algorithm 4, we can define robustness oracles for Linux-DRBG in Algorithm 5. Note that there are several differences from oracles in Algorithm 1 as follows.

- The refresh oracle is divided into  $\text{REF}_a$  and  $\text{REF}_f$  since Linux-DRBG accumulates the entropy gradually.
- If  $\text{REF}_f$  is invoked without sufficient entropy, it sets  $c$  to 0. Therefore, it is essential to supply sufficient entropy in a single  $\text{REF}_f$  call. This assumption is substantiated by the observation that, after system boot, Linux-DRBG invokes `crng_reseed` (equivalent to  $\text{REF}_f$ ) only if sufficient entropy is accumulated in its state. Also, after sufficient entropy is accumulated, Linux invokes `crng_reseed` every 60 seconds. After a sufficient amount of time has passed since the Linux system booted, it can be considered that 60 seconds is sufficient for accumulating enough entropy. Considering this behavior of Linux, even when a Seedless adversary attempts to leak the state of the DRBG, Linux DRBG can be assumed to accumulate sufficient entropy with a single invocation of  $\text{REF}_f$ . Therefore,  $\text{REF}_f$  can be modeled always to receive inputs with sufficient entropy.
- There are two ROR oracles,  $\text{ROR}_k$  and  $\text{ROR}_u$ .
- The ROR oracles do not work correctly if `ready` = 0, which means at least one  $\text{REF}_f$  call after entropy drain (will be defined in this section) is required.
- $\text{ROR}_k$  and  $\text{ROR}_u$  requires the number of CPU to generate random bits, and the process varies whether  $S.G.\text{flag}_{n_{cpu}}$  is 0 or 1.
- Since Linux-DRBG uses two cryptographic primitives, the weakly ideal cipher  $E$  and the random permutation  $\pi$ , and they allow inverse query, there exist four primitive query oracles for Linux-DRBG.  $E, E^{-1}, \pi, \pi^{-1}$  are that.

With the above definition, the procedure for Linux-DRBG Robust Game is described as follows.

**Linux-DRBG Robustness Game.**

1. Oracle runs `INIT()` procedure.
2. Adversary  $\mathcal{A}_2$  queries  $\text{ROR}_k, \text{ROR}_u, \text{GET}, \text{SET}, E, E^{-1}, \pi$  and  $\pi^{-1}$ .  $\mathcal{A}_1$  queries  $\text{REF}_a$  and  $\text{REF}_f$ . All query orders are free and can be done multiple times.
3.  $\mathcal{A}_2$  outputs  $b' \in \{0, 1\}$ , if  $b' = b$ ,  $\mathcal{A}$  wins.

**ENTROPY DRAIN.** We define entropy drains, which are events that make the DRBG lose its entropy by giving some information to the adversary. The following events are called **entropy drains**:

- Exactly after `INIT`,
- Calling oracles to `GET` or `SET`,
- Calling an oracle  $\text{ROR}_k$  or  $\text{ROR}_u$  when  $c < \lambda$  or `ready` = 0.

For convenient notation, we denote an entropy drain as ED from now on.

**CANONICAL ADVERSARY:** An adversary  $\mathcal{A}$  is called **canonical** if it follow the conditions below:

1.  $\mathcal{A}_2$  queries  $\text{ROR}_k$  or  $\text{ROR}_u$  only when  $c \geq \lambda$  and  $\text{ready} = 1$ .
2.  $\mathcal{A}_2$  queries  $\text{ROR}_k$  or  $\text{ROR}_u$ , only when the last construction query made by  $\mathcal{A}_1$  is  $\text{REF}_f$ .
3.  $\mathcal{A}_1$  does not query  $\text{REF}_a$  consecutively.
4. Between the last entropy drain and the first  $\text{ROR}_k$  or  $\text{ROR}_u$  query thereafter,  $\mathcal{A}_2$  does not query  $\text{GET}$  and  $\text{SET}$  in situations where  $c > 0$ .
5. Between the last entropy drain and the first  $\text{ROR}_k$  or  $\text{ROR}_u$  query thereafter,  $\mathcal{A}_1$  does not query  $\text{REF}_a$ .

We claim that we can assume the Linux-DRBG Robustness adversary  $\mathcal{A}$  is canonical. This assumption comes from the fact that the only difference between  $b = 0$  and  $b = 1$  is in  $\text{ROR}_k$  or  $\text{ROR}_u$  when  $c \geq \lambda$  and  $\text{ready} = 1$ , and  $\text{REF}_a$  only accumulate entropy, and  $\text{REF}_f$  is required to transfer the accumulated entropy to  $S.k_{\text{base}}$ , the state used in  $\text{ROR}_k$  or  $\text{ROR}_u$ . Therefore, for any adversary  $\mathcal{A}$  that violates the above condition, one can construct canonical adversary  $\mathcal{A}'$  using  $\mathcal{A}$  holding or simulating queries made by  $\mathcal{A}$  appropriately. The strategy of  $\mathcal{A}'$  is like below.

- If  $\mathcal{A}$  violates condition 1 or 4,  $\mathcal{A}'$  can easily simulate the query with primitive queries, because in that case the ideal world and real world behaviors are same.
- If  $\mathcal{A}$  violates one of condition 2,3,5,  $\mathcal{A}'$  just simply store  $\text{REF}_a$  queries after the last  $\text{REF}_f$  query. Then when  $\mathcal{A}$  queries  $\text{GET}$  or  $\text{REF}_f$ ,  $\mathcal{A}'$  can concatenate the queries into a  $\text{REF}_a$  or  $\text{REF}_f$ .

Therefore it is reasonable to assume the Linux-DRBG Robustness adversary  $\mathcal{A}$  is canonical.

## 5.2 Robustness Proof

The robustness advantage of the Linux DRBG is upper bounded in Theorem 1. In the statement of Theorem 1,  $\text{REF}$  (resp.  $\text{ROR}$ ) calls mean  $\text{REF}_f$  and  $\text{REF}_a$  (resp.  $\text{ROR}_k$  and  $\text{ROR}_u$ ) calls.

**Theorem 1.** *Let  $\mathcal{A}$  be a  $\lambda$ -legitimate robustness game adversary that makes  $p$  primitive query,  $q_1$   $\text{REF}$  query,  $q_2$   $\text{ROR}$  query,  $\ell_1$  maximum number of entropy input block in a single  $\text{REF}$  call,  $\ell_2$  maximum number of output block in a single  $\text{ROR}$  call,  $\sigma_1$  total number of entropy input blocks in every  $\text{REF}$ , and  $\sigma_2$  total number of output blocks in every  $\text{ROR}$ . Let  $\text{Adv}_{\text{rob}}(p, q_1, q_2, \ell_1, \ell_2, \sigma_1, \sigma_2, \lambda)$  be the advantage upper bound of all possible adversaries  $\mathcal{A}$ . If  $p + 3q_1 + 2\sigma_1 \leq 2^{n-1}$ ,*

the following inequality holds.

$$\begin{aligned}
& \mathbf{Adv}_{rob}(p, q_1, q_2, \ell_1, \ell_2, \sigma_1, \sigma_2, \lambda) \\
& \leq \frac{14q_1}{2^{0.5n}} + \frac{8q_2\ell_2(p + 2q_2 + \ell_2 + \sigma_2)}{2^{2n}} + \frac{8q_1(p + 3q_1 + \sigma_1)}{2^\lambda} \\
& + \frac{2p(p + 8q_2 + 27q_1 + 2\sigma_1) + 2q_1(72q_1 + 2\ell_1 + 31\sigma_1 + 2) + 4q_2(8q_2 + 4\sigma_2 + 1) + 4\sigma_1^2}{2^n} \\
& \leq \frac{14\sigma_1}{2^{0.5n}} + \frac{8\sigma_2^2(p + 4\sigma_2)}{2^{2n}} + \frac{8\sigma_1(p + 4\sigma_1)}{2^\lambda} \\
& + \frac{2p^2 + 2\sigma_1(29p + 107\sigma_1 + 2) + 4\sigma_2(4p + 12\sigma_2 + 1)}{2^n}.
\end{aligned}$$

Let  $S_0$  (resp.  $S_1$ ) be a system of ideal (resp. real) world robustness oracles. In the INIT in Algorithm 5, if  $b = 0$  (resp.  $b = 1$ ), then the system of oracles is  $S_0$  (resp.  $S_1$ ). Note that the only differences between  $S_0$  and  $S_1$  are the return values of oracles  $\text{ROR}_k$  and  $\text{ROR}_u$ . If in  $S_0$  (resp.  $S_1$ ), they return  $y_0$  (resp.  $y_1$ ) when  $c \geq \lambda$  and  $\text{ready} = 1$ .

**METHODOLOGY OF THE PROOF.** Our proof involves dividing the robustness distinguish game into subgames, proving the security of each, and then combining them. The subgames consist of the M-EXT game, which describes the distinguish game for REF calls, the  $\text{pREF}_a$  game, the  $\text{pREF}_f$  game, and the distinguish games for ROR calls, which include the  $\text{bROR}_k$  game,  $\text{bROR}_u$ ,  $\text{cROR}_k$  game, and  $\text{cROR}_u$  game. In the text, we first define the hybrid world  $S_h$  for convenience of proof, ensuring that the state updates uniformly randomly when accumulated entropy  $c \geq \lambda$  [14]. Subsequently, we define each subgame and its adversarial advantages, then claim Lemma 3 through game hopping with intermediate worlds that can apply each subgame, and prove the security of each subgame to prove Theorem 1 ultimately.

Among the subgames, the M-EXT game allows multiple M-EXT calls, differing from other subgames and previous proof methods that divided robustness into recovering security and preserving security [9,10,7,21]. Using the traditional method of splitting into several games with a single M-EXT call would result in each game's advantage having a  $p^2/2^n$  term, and when gathering these, a  $p^2q_1/2^n$  term would emerge, and we only could prove  $O(2^{n/3})$  security for Linux-DRBG. In contrast, the M-EXT game, by allowing multiple M-EXT calls, eliminates the need to gather security bound, leading to  $O(2^{n/2})$  security as shown in (2), and ultimately, we could prove that Linux-DRBG is secure up to  $O(\min(2^{n/2}, 2^{\lambda/2}))$  adversarial queries. We believe this technique could be applied to other DRBGs as well, potentially helping to raise their security upper bounds.

We denote  $\Delta_{\mathcal{A}}(S_0, S_1)$  for a seedless robustness distinguishing advantage of  $S_0$  and  $S_1$  for an adversary  $\mathcal{A}$  satisfying conditions in Theorem 1. Let  $S_h$  be a system of hybrid words that contains oracle  $\text{REF}_a^*$ ,  $\text{REF}_f^*$ ,  $\text{ROR}_k^*$  and  $\text{ROR}_u^*$  in Algorithm 6 instead of  $\text{REF}_a$ ,  $\text{REF}_f$ ,  $\text{ROR}_k$  and  $\text{ROR}_u$  in Algorithm 5. That means, when  $c \geq \lambda$ ,  $S_0$  outputs bit outputs randomly but updates its state with Linux-DRBG algorithm, and  $S_1$  outputs bit outputs and updates its state with

**Algorithm 6** Oracles for the hybrid world

<b>Procedure</b> $\text{REF}_a^*(I, \gamma)$ 1: $c \leftarrow c + \gamma$ //Update $c$ first. 2: <b>if</b> $c < \lambda$ <b>then</b> 3: $S \leftarrow \text{refresh}_a[E](S, I)$ 4: <b>else</b> 5: $S.h \leftarrow_{\$} \{0, 1\}^n$ 6: $S.t \leftarrow S.t + \text{len}$ 7: <b>return</b> $\gamma$	<b>Procedure</b> $\text{REF}_f^*(I, \gamma, I_{cpu})$ 1: $c \leftarrow c + \gamma$ //Update $c$ first. 2: <b>if</b> $c < \lambda$ <b>then</b> 3: $S \leftarrow \text{refresh}_f[E](S, I, I_{cpu})$ 4: <b>else</b> 5: $k_{\text{next}} \leftarrow_{\$} \{0, 1\}^n$ 6: $S.k_{\text{base}} \leftarrow_{\$} \{0, 1\}^n$ 7: $S.h \leftarrow \text{const}$ 8: $S.t \leftarrow 2n$ 9: $\text{ready} \leftarrow 1$ 10: $S.G\_flag_1 \leftarrow 1$ ; $S.G\_flag_2 \leftarrow 1$ ; $\dots$ ; $S.G\_flag_C \leftarrow 1$ 11: <b>return</b> $\gamma$
<b>Procedure</b> $\text{ROR}_k^*[\pi](\text{len}, n_{cpu})$ 1: <b>if</b> $c < \lambda$ or $\text{ready} = 0$ <b>then</b> 2: $(S, y) \leftarrow \text{next}_k[\pi](S, \text{len}, n_{cpu})$ 3: $c \leftarrow 0$ ; $\text{ready} \leftarrow 0$ 4: <b>else</b> 5: <b>if</b> $S.G\_flag_{n_{cpu}} = 1$ <b>then</b> 6: $S.k_{\text{base}} \leftarrow_{\$} \{0, 1\}^n$ 7: $S.G\_flag_{n_{cpu}} \leftarrow 0$ 8: $S.\text{ckey}_{n_{cpu}} \parallel y \leftarrow_{\$} \{0, 1\}^{n+\text{len}}$ 9: <b>return</b> $y$	<b>Procedure</b> $\text{ROR}_u^*[\pi](\text{len}, n_{cpu})$ 1: <b>if</b> $c < \lambda$ or $\text{ready} = 0$ <b>then</b> 2: $(S, y) \leftarrow \text{next}_u[\pi](S, \text{len}, n_{cpu})$ 3: $c \leftarrow 0$ ; $\text{ready} \leftarrow 0$ 4: <b>else</b> 5: <b>if</b> $S.G\_flag_{n_{cpu}} = 1$ <b>then</b> 6: $S.k_{\text{base}} \leftarrow_{\$} \{0, 1\}^n$ 7: $S.G\_flag_{n_{cpu}} \leftarrow 0$ 8: $S.\text{ckey}_{n_{cpu}} \parallel y \leftarrow_{\$} \{0, 1\}^{n+\text{len}}$ 9: <b>return</b> $y$

Linux-DRBG algorithm,  $S_h$  outputs bit outputs and updates its state randomly. Then, by the triangle inequality, the following holds:

$$\Delta_{\mathcal{A}}(S_0, S_1) \leq \Delta_{\mathcal{A}}(S_0, S_h) + \Delta_{\mathcal{A}}(S_h, S_1). \quad (1)$$

With (1), the following lemma holds.

**Lemma 2.** *In a distinguishing game,  $b \in \{0, 1\}$  is uniformly randomly chosen to select one of two worlds. For  $\Delta_{\mathcal{A}}(S_h, S_0)$ , if  $b = 0$  then  $S_h$  is selected. Otherwise,  $S_0$  is selected. For  $\Delta_{\mathcal{A}'}(S_h, S_1)$ , if  $b = 0$  then  $S_h$  is selected. Otherwise,  $S_1$  is selected. Then for any robustness adversary  $\mathcal{A}$ , there exists  $\mathcal{A}'$  that satisfies the following.*

$$\Delta_{\mathcal{A}}(S_h, S_0) \leq \Delta_{\mathcal{A}'}(S_h, S_1).$$

*Proof.* We can construct a distinguishing adversary  $\mathcal{A}'$  between  $S_h$  and  $S_1$  using an  $S_0$  and  $S_h$  distinguishing adversary  $\mathcal{A}$  as a subalgorithm.  $\mathcal{A}'$  passes oracle queries of  $\mathcal{A}$  to its oracles and just returns results to  $\mathcal{A}$  except when  $\mathcal{A}$  queries

ROR<sub>k</sub> or ROR<sub>u</sub> and conditions  $c \geq \lambda$  and  $\text{ready} = 1$  hold. If  $\mathcal{A}$  queries ROR<sub>k</sub> or ROR<sub>u</sub> and conditions  $c \geq \lambda$  and  $\text{ready} = 1$  hold, then  $\mathcal{A}'$  randomly picks a bitstring and returns it to  $\mathcal{A}$ . If  $b = 0$ , then  $\mathcal{A}'$  perfectly simulates  $S_h$  to  $\mathcal{A}$ , and if  $b = 1$ , then  $\mathcal{A}'$  perfectly simulates  $S_0$  to  $\mathcal{A}$ . Finally,  $\mathcal{A}'$  outputs  $b'$ , which is the final output of  $\mathcal{A}$ .

Hence, the following holds:

$$\Delta_{\mathcal{A}}(S_0, S_1) \leq 2\Delta_{\mathcal{A}'}(S_h, S_1).$$

Therefore, we only need to upper bound  $\Delta_{\mathcal{A}'}(S_h, S_1)$ .

**5.2.1 Games for Robustness Proof.** To upper bound  $\Delta_{\mathcal{A}'}(S_h, S_1)$ , we substitute oracles used in  $S_1$  to oracles used in  $S_h$  using the game hopping technique. We employ subgames for each substitution. The subgames are M-EXT game, pREF<sub>a</sub> game, pREF<sub>f</sub> game, bROR<sub>k</sub> game, bROR<sub>u</sub> game, cROR<sub>k</sub> game, cROR<sub>u</sub> game. The former 3 sub games are necessary to substitute REF<sub>a</sub> or REF<sub>f</sub> with REF<sub>a</sub><sup>\*</sup> or REF<sub>f</sub><sup>\*</sup>, the latter 4 sub games are necessary to substitute ROR<sub>k</sub> or ROR<sub>u</sub> with ROR<sub>k</sub> or ROR<sub>u</sub><sup>\*</sup>. One can see how the following subgames are used to prove Theorem 1 via Lemma 3.

With oracles in Algorithm 7, M-EXT game, pREF<sub>a</sub> game, pREF<sub>f</sub> game processes are defined like below.

**M-EXT game.**

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $E, E^{-1}$  and  $\mathcal{A}_1$  queries M-EXT[ $E$ ]( $inc, h_1, I, I_{cpu}$ ) multiple times, and gets the output.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

**pREF<sub>a</sub> game.**

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $E, E^{-1}$  multiple time and  $\mathcal{A}_1$  queries pREF<sub>a</sub>[ $E$ ]( $I$ ) once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

**pREF<sub>f</sub> game.**

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $E, E^{-1}$  multiple time and  $\mathcal{A}_1$  queries pREF<sub>f</sub>[ $E$ ]( $I, I_{cpu}$ ) once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

**Algorithm 7** Oracles for refresh sub Games**Procedure** INIT()

- 1:  $b \leftarrow_{\$} \{0, 1\}$
- 2:  $E \leftarrow_{\$} \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$
- 3:  $k \leftarrow_{\$} \{0, 1\}^n$  //No usage in M-EXT game

**Procedure** M-EXT[ $E$ ]( $inc, h_1, I, I_{cpu}$ )

- 1: **if**  $b = 0$  **then**
- 2:    $s \leftarrow_{\$} \{0, 1\}^{2n}$
- 3: **else**
- 4:    $k \leftarrow \text{COMP}[E](inc, h_1, I)$
- 5:    $I_{cpu} \leftarrow 0^n \parallel k \parallel I_{cpu} \parallel 0^{c-1}$
- 6:    $s_L \leftarrow \text{COMP}[E](0, const, I_{cpu} \parallel 0)$
- 7:    $s_R \leftarrow \text{COMP}[E](0, const, I_{cpu} \parallel 1)$
- 8:    $s \leftarrow s_L \parallel s_R$
- 9: **return** ( $inc, h_1, s$ )

**Procedure** pREF<sub>a</sub>[ $E$ ]( $I$ )

- 1: **if**  $b = 0$  **then**
- 2:    $h \leftarrow_{\$} \{0, 1\}^n$
- 3: **else**
- 4:    $I_0 \leftarrow 0^n \parallel k$
- 5:    $(I_1, \dots, I_\ell) \xleftarrow{2n} I$
- 6:    $h \leftarrow const$
- 7:   **for**  $i \leftarrow 0$  **to**  $\ell$  **do**
- 8:      $h \leftarrow B[E](h, (i + 1) \cdot 2n, I_i)$
- 9: **return**  $h$

**Procedure** pREF<sub>f</sub>[ $E$ ]( $I, I_{cpu}$ )

- 1: **if**  $b = 0$  **then**
- 2:    $s \leftarrow_{\$} \{0, 1\}^{2n}$
- 3: **else**
- 4:    $y \leftarrow k$
- 5:    $y \leftarrow \text{COMP}[E](0, const, 0^n \parallel y \parallel I)$
- 6:    $I_{cpu} \leftarrow 0^n \parallel y \parallel I_{cpu} \parallel 0^{c-1}$
- 7:    $s_L \leftarrow \text{COMP}[E](0, const, I_{cpu} \parallel 0)$
- 8:    $s_R \leftarrow \text{COMP}[E](0, const, I_{cpu} \parallel 1)$
- 9:    $s \leftarrow s_L \parallel s_R$
- 10: **return**  $s$

And define some values like below.

- $\text{Adv}_{\text{M-EXT}}(p, q, \sigma, \lambda)$  : The advantage upper bound against any  $\lambda$ -legitimate adversary  $\mathcal{A}$  that makes at most  $p$  queries to  $E$  or  $E^{-1}$ ,  $q$  queries to M-EXT, and the total length of entropy input  $I$  is less than  $2n\sigma$  bits.
- $\text{Adv}_{\text{pREF}_a}(p, \ell)$  : The advantage upper bound against any adversary  $\mathcal{A}$  that makes at most  $p$  queries to  $E$  or  $E^{-1}$ , entropy input  $I$ 's length for pREF<sub>a</sub> is less than  $2n\ell$  bits.
- $\text{Adv}_{\text{pREF}_f}(p, \ell)$  : The advantage upper bound against any adversary  $\mathcal{A}$  that makes at most  $p$  queries to  $E$  or  $E^{-1}$ ,  $r$  entropy input blocks to pREF<sub>f</sub>, and the input block  $I$  and  $I_{cpu}$ 's length is less than  $2n\ell$  bits.

**Algorithm 8** Oracles for Base ROR subgames

---

**Procedure** INIT()

- 1:  $k_{\text{base}} \leftarrow_{\$} \{0, 1\}^n$
- 2:  $b \leftarrow_{\$} \{0, 1\}$
- 3:  $\pi \leftarrow_{\$} \text{Perm}(2n)$

<p><b>Procedure</b> bROR<sub>k</sub>[<math>\pi</math>](<math>len</math>)</p> <ol style="list-style-type: none"> <li>1: <b>if</b> <math>b = 0</math> <b>then</b></li> <li>2:   <math>y_0 \leftarrow_{\\$} \{0, 1\}^{len+2n}</math></li> <li>3:   <b>return</b> <math>y_0</math></li> <li>4: <b>else</b></li> <li>5:   <math>k_{\text{base}} \parallel c\_key \leftarrow \text{CB}[\pi](k_{\text{base}}, 0, 2n)</math></li> <li>6:   <math>k \leftarrow c\_key</math></li> <li>7:   <math>c\_key \parallel y_1 \leftarrow \text{CB}[\pi](c\_key, 0, 2n)</math></li> <li>8:   <math>B \leftarrow \text{CB}[\pi](k, 1, len - n)</math></li> <li>9:   <math>y_1 \leftarrow y_1 \parallel B</math></li> <li>10:   <math>y_1 \leftarrow y_1[0 : len - 1]</math></li> <li>11:   <b>return</b> <math>k_{\text{base}} \parallel c\_key \parallel y_1</math></li> </ol>	<p><b>Procedure</b> bROR<sub>u</sub>[<math>\pi</math>](<math>len</math>)</p> <ol style="list-style-type: none"> <li>1: <b>if</b> <math>b = 0</math> <b>then</b></li> <li>2:   <math>y_0 \leftarrow_{\\$} \{0, 1\}^{len+2n}</math></li> <li>3:   <b>return</b> <math>y_0</math></li> <li>4: <b>else</b></li> <li>5:   <math>k_{\text{base}} \parallel c\_key \leftarrow \text{CB}[\pi](k_{\text{base}}, 0, 2n)</math></li> <li>6:   <math>c\_key \parallel k \leftarrow \text{CB}[\pi](c\_key, 0, 2n)</math></li> <li>7:   <b>if</b> <math>len \leq n</math> <b>then</b></li> <li>8:     <b>return</b> <math>k_{\text{base}} \parallel c\_key \parallel k[0 : len - 1]</math></li> <li>9:   <math>y_1 \leftarrow \text{CB}[\pi](k, 1, len)</math></li> <li>10:   <math>y_1 \leftarrow y_1[0 : len - 1]</math></li> <li>11:   <b>return</b> <math>k_{\text{base}} \parallel c\_key \parallel y_1</math></li> </ol>
--	--

---

**Algorithm 9** Oracles for CPU ROR subgames

---

**Procedure** INIT()

- 1:  $c\_key \leftarrow_{\$} \{0, 1\}^n$
- 2:  $b \leftarrow_{\$} \{0, 1\}$
- 3:  $\pi \leftarrow_{\$} \text{Perm}(2n)$

<p><b>Procedure</b> cROR<sub>k</sub>[<math>\pi</math>](<math>len</math>)</p> <ol style="list-style-type: none"> <li>1: <b>if</b> <math>b = 0</math> <b>then</b></li> <li>2:   <math>y_0 \leftarrow_{\\$} \{0, 1\}^{len+2n}</math></li> <li>3:   <b>return</b> <math>y_0</math></li> <li>4: <b>else</b></li> <li>5:   <math>k \leftarrow c\_key</math></li> <li>6:   <math>c\_key \parallel y_1 \leftarrow \text{CB}[\pi](c\_key, 0, 2n)</math></li> <li>7:   <math>B \leftarrow \text{CB}[\pi](k, 1, len - n)</math></li> <li>8:   <math>y_1 \leftarrow y_1 \parallel B</math></li> <li>9:   <math>y_1 \leftarrow y_1[0 : len - 1]</math></li> <li>10:   <b>return</b> <math>k_{\text{base}} \parallel c\_key \parallel y_1</math></li> </ol>	<p><b>Procedure</b> cROR<sub>u</sub>[<math>\pi</math>](<math>len</math>)</p> <ol style="list-style-type: none"> <li>1: <b>if</b> <math>b = 0</math> <b>then</b></li> <li>2:   <math>y_0 \leftarrow_{\\$} \{0, 1\}^{len+2n}</math></li> <li>3:   <b>return</b> <math>y_0</math></li> <li>4: <b>else</b></li> <li>5:   <math>c\_key \parallel k \leftarrow \text{CB}[\pi](c\_key, 0, 2n)</math></li> <li>6:   <b>if</b> <math>len \leq n</math> <b>then</b></li> <li>7:     <b>return</b> <math>k_{\text{base}} \parallel c\_key \parallel k[0 : len - 1]</math></li> <li>8:   <math>y_1 \leftarrow \text{CB}[\pi](k, 1, len)</math></li> <li>9:   <math>y_1 \leftarrow y_1[0 : len - 1]</math></li> <li>10:   <b>return</b> <math>k_{\text{base}} \parallel c\_key \parallel y_1</math></li> </ol>
---	---

---

With oracles in Algorithm 8 and in Algorithm 9, we can define bROR<sub>k</sub> game, bROR<sub>u</sub> and cROR<sub>k</sub> game, cROR<sub>u</sub> game processes like below.



**bROR<sub>x</sub> game.** ( $x \in \{k, u\}$ )

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $\pi, \pi^{-1}$  multiple time and queries  $\text{bROR}_x[\pi](len)$  once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

**cROR<sub>x</sub> game.** ( $x \in \{k, u\}$ )

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $\pi, \pi^{-1}$  multiple time and queries  $\text{cROR}_x[\pi](len)$  once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

And for all  $\mathcal{O} \in \{\text{bROR}_k, \text{bROR}_u, \text{cROR}_k, \text{cROR}_u\}$ , let  $\mathbf{Adv}_{\mathcal{O}}(p, \ell)$  be the advantage upper bound against any adversary  $\mathcal{A}$  that makes at most  $p$  queries to  $\pi$  or  $\pi^{-1}$ , inputs  $2n\ell$  to  $\mathcal{O}$ .

After hopping every game, we obtain the upper bound of  $\Delta_{\mathcal{A}}(S_h, S_1)$ . The result is presented in the Lemma 3.

**Lemma 3.** *For any  $\lambda$ -legitimate robustness adversary  $\mathcal{A}$  satisfying conditions in Theorem 1, the following holds.*

$$\begin{aligned}
\Delta_{\mathcal{A}}(S_0, S_1) &\leq 2\Delta_{\mathcal{A}}(S_h, S_1) \\
&\leq 2\mathbf{Adv}_{\text{M-EXT}}(p + 3q_1 + \sigma_1, q_1, \sigma_1, \lambda) \\
&\quad + 2q_1 (\mathbf{Adv}_{\text{pREF}_a}(p + 3q_1 + \sigma_1, \ell_1) + \mathbf{Adv}_{\text{pREF}_f}(p + 3q_1 + \sigma_1, \ell_1)) \\
&\quad + 2q_2 (\mathbf{Adv}_{\text{bROR}_k}(p + 2q_2 + \sigma_2, \ell_2) + \mathbf{Adv}_{\text{cROR}_k}(p + 2q_2 + \sigma_2, \ell_2)) \\
&\quad + 2q_2 (\mathbf{Adv}_{\text{bROR}_u}(p + 2q_2 + \sigma_2, \ell_2) + \mathbf{Adv}_{\text{cROR}_u}(p + 2q_2 + \sigma_2, \ell_2)).
\end{aligned}$$

The proof of Lemma 3 is deferred to Section 5.3. Now, it remains to calculate the advantage of the subgames introduced in Lemma 3. This is summarized in Lemma 4.

**Lemma 4.** *If  $p + \sigma \leq 2^{n-1}$  and  $p + \ell \leq 2^{n-1}$ , the following inequalities hold:*

$$\mathbf{Adv}_{\text{M-EXT}}(p, q, \sigma, \lambda) \leq \frac{9pq + 4q\sigma + \sigma^2}{2^n} + \frac{3q}{2^{0.5n}} + \frac{p^2}{2^n} + \frac{4pq}{2^\lambda}, \quad (2)$$

$$\mathbf{Adv}_{\text{pREF}_a}(p, \ell) \leq \frac{3p}{2^n} + \frac{\ell}{2^n} + \frac{1}{2^{0.5n}}, \quad (3)$$

$$\mathbf{Adv}_{\text{pREF}_f}(p, \ell) \leq \frac{9p}{2^n} + \frac{\ell + 2}{2^n} + \frac{3}{2^{0.5n}}, \quad (4)$$

$$\mathbf{Adv}_{\text{bROR}_k}(p, \ell) \leq \frac{1 + 2p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}, \quad (5)$$

$$\mathbf{Adv}_{\text{cROR}_k}(p, \ell) \leq \frac{p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}, \quad (6)$$

$$\mathbf{Adv}_{\text{bROR}_u}(p, \ell) \leq \frac{1 + 3p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}, \quad (7)$$

$$\mathbf{Adv}_{\text{cROR}_u}(p, \ell) \leq \frac{2p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}. \quad (8)$$

The proof of Lemma 4 is deferred to Section 5.4.

### 5.3 Proof of Lemma 3

To upper bound  $\Delta_{\mathcal{A}}(S_h, S_1)$ , we substitute oracles used in  $S_1$  to oracles used in  $S_h$  using the game hopping technique. To substitute the first  $\text{REF}_f$  query after  $ED$ , we define a system  $R$ . Define  $R$  as a system based on  $S_1$ . The only difference between  $R$  and  $S_1$  is that the first  $\text{REF}_f$  query after  $ED$  is  $\text{REF}_f^*$  in  $R$ . By the triangle inequality, the following holds:

$$\Delta_{\mathcal{A}}(S_h, S_1) \leq \Delta_{\mathcal{A}}(S_h, R) + \Delta_{\mathcal{A}}(R, S_1). \quad (9)$$

To upper bound  $\Delta_{\mathcal{A}}(S_h, S_1)$  in (9), we first upper bound  $\Delta_{\mathcal{A}}(R, S_1)$  in Lemma 5.

**Lemma 5.**  $\Delta_{\mathcal{A}}(R, S_1) \leq \mathbf{Adv}_{\text{M-EXT}}(p + 3q_1 + \sigma_1, r, \sigma_1, \lambda)$ .

*Proof.* M-EXT game adversary  $\mathcal{A}' = (\mathcal{A}'_1, \mathcal{A}'_2)$  can be constructed using  $R$  and  $S_1$  distinguishing adversary  $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$  by answering queries of  $\mathcal{A}$  as follows. First  $\mathcal{A}'$  get access to oracles for  $E$  and  $E^{-1}$  from M-EXT game, and then prepares for simulation as follows:

–  $c \leftarrow 0$ ,  $\text{ready} \leftarrow 0$ ,  $\pi \leftarrow_{\$} \text{Perm}(2n)$ ,  $S \leftarrow \text{setup}()$ .

Then,  $\mathcal{A}'$  handles queries of  $\mathcal{A}$  as follows:

–  $\text{REF}_f[E](I, \gamma, I_{\text{cpu}})$ : If this query is the first  $\text{REF}_f$  query after  $ED$ , then  $\mathcal{A}'_1$  first checks  $S.k_{\text{next}} \neq \epsilon$ . If  $S.k_{\text{next}} \neq \epsilon$ , then  $\mathcal{A}'_1$  sets  $I$  as follows:

$$I \leftarrow 0^n \parallel S.k_{\text{next}} \parallel I.$$

After considering  $S.k_{\text{next}}$ ,  $\mathcal{A}'_1$  queries  $(S.t, S.h, I, I_{\text{cpu}})$  to the M-EXT. Then  $\mathcal{A}'_2$  gets its return value  $(\text{inc}, h_1, s = s_L \parallel s_R)$ . Then,  $\mathcal{A}'$  updates

- $S.k_{\text{next}} \leftarrow s_L, S.h \leftarrow \text{const}, S.t \leftarrow 2n, S.k_{\text{base}} \leftarrow s_R,$
  - $S.G\_flag_1 \leftarrow 1, S.G\_flag_2 \leftarrow 1, \dots, S.G\_flag_C \leftarrow 1.$
- $\mathcal{A}'_2$  gives estimated entropy  $\gamma$  to  $\mathcal{A}_2$ . Finally,  $\mathcal{A}'$  sets  $c \leftarrow c + \gamma$  and sets  $\text{ready} \leftarrow 1$ . Note that  $\mathcal{A}'_1$  does not give more information to  $\mathcal{A}'_2$  than  $\mathcal{A}_1$  gives to  $\mathcal{A}_2$ . Therefore, if  $\mathcal{A}_1$  is legitimate then  $\mathcal{A}'_1$  is legitimate.
- E and  $E^{-1}$ :  $\mathcal{A}'$  queries to the E and  $E^{-1}$  oracles of the M-EXT game. Then returns the value to  $\mathcal{A}$ .
  - Other queries:  $\mathcal{A}'$  properly simulates the queries. Note that  $\mathcal{A}'$  needs to query  $E$  or  $E^{-1}$  at most  $3q_1 + \sigma_1$  times to simulate  $\text{REF}_a$  and  $\text{REF}_f$  itself.

Finally,  $\mathcal{A}'$  outputs the final output of  $\mathcal{A}$ . If a random coin  $b$  of the M-EXT game is 0, then  $\mathcal{A}'$  simulates  $R$  to  $\mathcal{A}$ . Otherwise  $\mathcal{A}'$  simulates  $S_1$  to  $\mathcal{A}$ .

To upper bound  $\Delta_{\mathcal{A}}(S_h, R)$  in (9), we define a system  $T_i$  for  $i = 0, \dots, q_1$ . The system  $T_0$  is equal to  $R$ . The system  $T_{i+1}$  is defined based on  $T_i$ . The only difference between  $T_i$  and  $T_{i+1}$  is that  $T_{i+1}$  uses  $\text{REF}_f^*$  instead of  $(i+1)$ -th  $\text{REF}_f$  query. Then the following holds by the triangle inequality:

$$\Delta_{\mathcal{A}}(S_h, R) \leq \sum_{i=0}^{q_1-1} \Delta_{\mathcal{A}}(T_i, T_{i+1}) + \Delta_{\mathcal{A}}(S_h, T_{q_1}). \quad (10)$$

To derive an upper bound of  $\Delta_{\mathcal{A}}(S_h, R)$  in (10), we first upper bound  $\Delta_{\mathcal{A}}(T_i, T_{i+1})$  for  $i = 0, \dots, q_1 - 1$  in Lemma 6.

**Lemma 6.** *For all  $i = 0, \dots, q_1 - 1$ , the following holds:*

$$\Delta_{\mathcal{A}}(T_i, T_{i+1}) \leq \text{Adv}_{\text{pREF}_f}(p + 3q_1 + \sigma_1, \ell_1).$$

*Proof.* We construct  $\text{pREF}_f$  adversary  $\mathcal{A}'$  as follows. First  $\mathcal{A}'$  get access to oracles for  $E$  and  $E^{-1}$  from  $\text{pREF}_f$  game, and then prepares for simulation as follows:

- $c \leftarrow 0, \text{ready} \leftarrow 0, \pi \leftarrow_{\$} \text{Perm}(2n), S \leftarrow \text{setup}().$

Then,  $\mathcal{A}'$  handles queries of  $\mathcal{A}$  as follows:

- $\text{REF}_f[E](I, \gamma, I_{\text{cpu}})$ : If this query is the  $(i+1)$ -th  $\text{REF}_f$ , then  $\mathcal{A}'_1$  first checks  $S.k_{\text{next}} \neq \epsilon$ . If  $S.k_{\text{next}} \neq \epsilon$ , then  $\mathcal{A}'_1$  sets  $I$  as follows:

$$I \leftarrow 0^n \parallel S.k_{\text{next}} \parallel I.$$

After considering  $S.k_{\text{next}}$ ,  $\mathcal{A}'_1$  queries  $(I, I_{\text{cpu}})$  to the  $\text{pREF}_f$ . Then  $\mathcal{A}'_2$  gets its return value  $s = s_L \parallel s_R$ . Then,  $\mathcal{A}'$  updates

- $S.k_{\text{next}} \leftarrow s_L, S.h \leftarrow \text{const}, S.t \leftarrow 2n, S.k_{\text{base}} \leftarrow s_R,$
  - $S.G\_flag_1 \leftarrow 1, S.G\_flag_2 \leftarrow 1, \dots, S.G\_flag_C \leftarrow 1.$
- $\mathcal{A}'_2$  gives estimated entropy  $\gamma$  to  $\mathcal{A}_2$ . Finally,  $\mathcal{A}'$  sets  $c \leftarrow c + \gamma$ .
- E and  $E^{-1}$ :  $\mathcal{A}'$  queries to the E and  $E^{-1}$  oracles of the  $\text{pREF}_f$  game. Then returns the value to  $\mathcal{A}$ .
  - Other queries:  $\mathcal{A}'$  properly simulates the queries. Note that  $\mathcal{A}'$  needs to query  $E$  or  $E^{-1}$  at most  $3q_1 + \sigma_1$  times to simulate  $\text{REF}_a$  and  $\text{REF}_f$  itself.

Finally,  $\mathcal{A}'$  outputs the final output of  $\mathcal{A}$ . If a random coin  $b$  of the  $\text{pREF}_f$  game is 0, then  $\mathcal{A}'$  simulates  $T_{i+1}$  to  $\mathcal{A}$ . Otherwise  $\mathcal{A}'$  simulates  $T_i$  to  $\mathcal{A}$ . Hence, the following holds:

$$\Delta_{\mathcal{A}}(T_i, T_{i+1}) \leq \mathbf{Adv}_{\text{pREF}_f}(p + 3q_1 + \sigma_1, \ell_1).$$

To upper bound  $\Delta_{\mathcal{A}}(S_h, T_r)$  in (10), we define a system  $W_i$  for  $i = 0, \dots, q_1$ . The system  $W_0$  is equal to  $T_r$ . The system  $W_{i+1}$  is defined based on  $W_i$ . The only difference between  $W_i$  and  $W_{i+1}$  is that  $W_{i+1}$  uses  $\text{REF}_a^*$  instead of  $(i+1)$ -th  $\text{REF}_a$  query. Then the following holds by the triangle inequality:

$$\Delta_{\mathcal{A}}(S_h, T_r) \leq \sum_{i=0}^{r-1} \Delta_{\mathcal{A}}(W_i, W_{i+1}) + \Delta_{\mathcal{A}}(S_h, W_r). \quad (11)$$

To derive an upper bound of  $\Delta_{\mathcal{A}}(S_h, T_r)$  in (11), we upper bound  $\Delta_{\mathcal{A}}(W_i, W_{i+1})$  for  $i = 0, \dots, q_1 - 1$  in Lemma 7.

**Lemma 7.** *For all  $i = 0, \dots, q_1 - 1$ , the following holds:*

$$\Delta_{\mathcal{A}}(W_i, W_{i+1}) \leq \mathbf{Adv}_{\text{pREF}_a}(p + 3q_1 + \sigma_1, \ell_1).$$

*Proof.* We construct  $\text{pREF}_a$  adversary  $\mathcal{A}'$  as follows. First  $\mathcal{A}'$  gets access to oracles for  $E$  and  $E^{-1}$  from  $\text{pREF}_a$  game, and then prepares for simulation as follows:

- $c \leftarrow 0$ ,  $\text{ready} \leftarrow 0$ ,  $\pi \leftarrow_{\$} \text{Perm}(2n)$ ,  $S \leftarrow \text{setup}()$ .

Then,  $\mathcal{A}'$  handles queries of  $\mathcal{A}$  as follows:

- $\text{REF}_a[E](I, \gamma)$ : If this query is the  $(i+1)$ -th  $\text{REF}$ , then  $\mathcal{A}'_1$  first checks  $S.k_{\text{next}} \neq \epsilon$ . If  $S.k_{\text{next}} \neq \epsilon$ , then  $\mathcal{A}'_1$  sets  $I$  as follows:

$$I \leftarrow 0^n \parallel S.k_{\text{next}} \parallel I.$$

After considering  $S.k_{\text{next}}$ ,  $\mathcal{A}'_1$  queries  $I$  to the  $\text{pREF}_a$ . Then  $\mathcal{A}'_2$  gets its return value  $h$ .  $\mathcal{A}'$  calculates  $\ell$  which is block length of  $I$  as the following:

$$(I_1, \dots, I_\ell) \xleftarrow{2n} I.$$

Then,  $\mathcal{A}'$  updates

- $S.h \leftarrow h$ ,  $S.t \leftarrow S.t + (\ell + 2) \cdot 2n$ .
- $\mathcal{A}'_2$  gives estimated entropy  $\gamma$  to  $\mathcal{A}_2$ . Finally,  $\mathcal{A}'$  sets  $c \leftarrow c + \gamma$ .
- $E$  and  $E^{-1}$ :  $\mathcal{A}'$  queries to the  $E$  and  $E^{-1}$  oracles of the  $\text{pREF}_a$  game. Then returns the value to  $\mathcal{A}$ .
- Other queries:  $\mathcal{A}'$  properly simulates the queries. Note that  $\mathcal{A}'$  needs to query  $E$  or  $E^{-1}$  at most  $3q_1 + \sigma_1$  times to simulate  $\text{REF}_a$  and  $\text{REF}_f$  itself.

Finally,  $\mathcal{A}'$  outputs the final output of  $\mathcal{A}$ . If a random coin  $b$  of the  $\text{pREF}_a$  game is 0, then  $\mathcal{A}'$  simulates  $W_{i+1}$  to  $\mathcal{A}$ . Otherwise  $\mathcal{A}'$  simulates  $W_i$  to  $\mathcal{A}$ . Hence, the following holds:

$$\Delta_{\mathcal{A}}(W_i, W_{i+1}) \leq \mathbf{Adv}_{\text{pREF}_a}(p + 3q_1 + \sigma_1, \ell_1)$$

Now we need to derive an upper bound of  $\Delta_{\mathcal{A}}(W_r, S_h)$  in (11). Note that the differences between the two systems are usage of ROR or ROR\*. To substitute ROR to ROR\*, we define a system  $H_i^C$  for  $i = 0, \dots, q_2$ . Let  $H_0^C = W_r$ . Because Linux DRBG uses a multi-core system, we need to define worlds as follows:

- $H_i^{Bn}$  : if  $i$ -th ROR is  $\text{ROR}_k[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 1$ , then it runs  $\text{ROR}_k^*[\pi](len, n_{cpu})$ . Otherwise, it works like  $H_{i-1}^C$ .
- $H_i^B$  : if  $i$ -th ROR is  $\text{ROR}_u[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 1$ , then it runs  $\text{ROR}_u^*[\pi](len, n_{cpu})$ . Otherwise, it works like  $H_i^{Bn}$ .
- $H_i^{Cn}$  : if  $i$ -th ROR is  $\text{ROR}_k[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 0$ , then it runs  $\text{ROR}_k^*[\pi](len, n_{cpu})$ . Otherwise, it works like  $H_i^B$ .
- $H_i^C$  : if  $i$ -th ROR is  $\text{ROR}_u[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 0$ , then it runs  $\text{ROR}_u^*[\pi](len, n_{cpu})$ . Otherwise, it works like  $H_i^{Cn}$ .

Then the following holds by the triangle inequality:

$$\Delta_{\mathcal{A}}(S_h, W_r) \leq \sum_{i=0}^{q-1} \Delta_{\mathcal{A}}(H_i^C, H_{i+1}^C). \quad (12)$$

Note that  $H_q^C = S_h$ .

To bound  $\Delta_{\mathcal{A}}(H_i^C, H_{i+1}^C)$ , we use the triangle inequality as follows:

$$\begin{aligned} \Delta_{\mathcal{A}}(H_i^C, H_{i+1}^C) &\leq \Delta_{\mathcal{A}}(H_i^C, H_{i+1}^{Bn}) + \Delta_{\mathcal{A}}(H_{i+1}^{Bn}, H_{i+1}^B) \\ &\quad + \Delta_{\mathcal{A}}(H_{i+1}^B, H_{i+1}^{Cn}) + \Delta_{\mathcal{A}}(H_{i+1}^{Cn}, H_{i+1}^C). \end{aligned}$$

First, we upper bound  $\Delta_{\mathcal{A}}(H_i^C, H_{i+1}^{Bn})$ .

**Lemma 8.** *For  $i = 0, \dots, q_2 - 1$ , the following holds:*

$$\Delta_{\mathcal{A}}(H_i^C, H_{i+1}^{Bn}) \leq \text{Adv}_{\text{bROR}_k}(p + 2q_2 + \sigma_2, \ell_2).$$

*Proof.*  $H_i^C$  and  $H_{i+1}^{Bn}$  have different behaviors only when the  $(i+1)$ -th ROR is  $\text{ROR}_k[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 1$ . So we assume  $(i+1)$ -th ROR is  $\text{ROR}_k[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 1$ . We can construct  $\text{bROR}_k$  game adversary  $\mathcal{A}'$  using  $H_i^C$  and  $H_{i+1}^{Bn}$  distinguishing adversary  $\mathcal{A}$  as follows. First,  $\mathcal{A}'$  get access to oracles for  $\pi$  and  $\pi^{-1}$  from  $\text{bROR}_k$  game, and then prepares for simulation as follows:

- $c \leftarrow 0$ ,  $\text{ready} \leftarrow 0$ ,  $E \leftarrow_{\$} \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$ ,  $S \leftarrow \text{setup}()$ .

then  $\mathcal{A}'$  handles queries of  $\mathcal{A}$  as follows:

- $(i+1)$ -th  $\text{ROR}_k[\pi](len, n_{cpu})$ :  $\mathcal{A}'$  queries  $len$  to  $\text{bROR}_k$  game and gets its return value  $y$ .  $\mathcal{A}'$  partitions  $y$  as follows.

$$k_{\text{base}} || c.\text{key} || y^* \leftarrow y.$$

$\mathcal{A}'$  updates the state  $S$  as  $S.k_{\text{base}} \leftarrow k_{\text{base}}$  and  $S.\text{key}_{n_{cpu}} \leftarrow c.\text{key}$ . Then give  $y^*$  to  $\mathcal{A}$ .

- $\pi$  and  $\pi^{-1}$ :  $\mathcal{A}'$  queries to the  $\pi$  and  $\pi^{-1}$  oracles of the  $\text{bROR}_k$  game. Then returns the value to  $\mathcal{A}$ .
- Other queries:  $\mathcal{A}'$  simulates the query to  $\mathcal{A}$ . Note that  $\mathcal{A}'$  needs to query  $\pi$  or  $\pi^{-1}$  at most  $2q_2 + \sigma_2$  times to simulate  $\text{ROR}_k$  and  $\text{ROR}_u$  itself.

Finally,  $\mathcal{A}'$  outputs the final output of  $\mathcal{A}$ . If a random coin  $b$  of  $\text{bROR}_k$  game is 0, then  $\mathcal{A}'$  simulates  $H_{i+1}^{Bn}$  to  $\mathcal{A}$ . Otherwise,  $\mathcal{A}'$  simulates  $H_i^C$  to  $\mathcal{A}$ .

Now we upper bound  $\Delta_{\mathcal{A}}(H_i^{Bn}, H_i^B)$ .

**Lemma 9.**  $\Delta_{\mathcal{A}}(H_i^{Bn}, H_i^B) \leq \mathbf{Adv}_{\text{bROR}_u}(p + 2q_2 + \sigma_2, \ell_2)$ .

*Proof.*  $H_i^B$  and  $H_i^{Bn}$  have different behaviors only when the  $i$ -th  $\text{ROR}$  is  $\text{ROR}_u[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 1$ . So we assume  $i$ -th  $\text{ROR}$  is the case. Then this lemma can be proved similarly to the Lemma 8. The only difference is for  $i$ -th  $\text{ROR}_u$  call,  $\mathcal{A}'$  queries to  $\text{bROR}_u$  instead of  $\text{bROR}_k$ .

Now we upper bound  $\Delta_{\mathcal{A}}(H_i^B, H_i^{Cn})$ .

**Lemma 10.**  $\Delta_{\mathcal{A}}(H_i^B, H_i^{Cn}) \leq \mathbf{Adv}_{\text{cROR}_k}(p + 2q_2 + \sigma_2, \ell_2)$ .

*Proof.*  $H_i^B$  and  $H_i^{Cn}$  have different behaviors only when  $i$ -th  $\text{ROR}$  call is  $\text{ROR}_k[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 0$ . So we assume  $i$ -th  $\text{ROR}$  call is the case. We can construct  $\text{cROR}_k$  game adversary  $\mathcal{A}'$  using  $H_i^B$  and  $H_i^{Cn}$  distinguishing adversary  $\mathcal{A}$  as follows. First,  $\mathcal{A}'$  get access to oracles for  $\pi$  and  $\pi^{-1}$  from  $\text{cROR}_k$  game, and then prepares for simulation as follows:

- $c \leftarrow 0$ ,  $\text{ready} \leftarrow 0$ ,  $E \leftarrow_{\$} \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$ ,  $S \leftarrow \text{setup}()$ .

then  $\mathcal{A}'$  handles queries of  $\mathcal{A}$  as follows:

- $i$ -th  $\text{ROR}_k[\pi](len, n_{cpu})$ :  $\mathcal{A}'$  queries  $len$  to  $\text{cROR}_k$  and gets its return value  $y$ .  $\mathcal{A}'$  partitions  $y$  as follows.

$$\text{c.key} || y^* \leftarrow y.$$

$\mathcal{A}'$  updates the state  $S$  as  $S.\text{ckey}_{n_{cpu}} \leftarrow \text{c.key}$ . Then give  $y^*$  to  $\mathcal{A}$ .

- $\pi$  and  $\pi^{-1}$ :  $\mathcal{A}'$  queries to the  $\pi$  and  $\pi^{-1}$  oracles of the  $\text{cROR}_k$  game. Then returns the value to  $\mathcal{A}$ .
- Other queries:  $\mathcal{A}'$  simulates the query to  $\mathcal{A}$ . Note that  $\mathcal{A}'$  needs to query  $\pi$  or  $\pi^{-1}$  at most  $2q_2 + \sigma_2$  times to simulate  $\text{ROR}_k$  and  $\text{ROR}_u$  itself.

Finally,  $\mathcal{A}'$  outputs the final output of  $\mathcal{A}$ . If a random coin  $b$  of  $\text{cROR}_k$  game is 0, then  $\mathcal{A}'$  simulates  $H_i^{Cn}$  to  $\mathcal{A}$ . Otherwise,  $\mathcal{A}'$  simulates  $H_i^B$  to  $\mathcal{A}$ .

Finally, we upper bound  $\Delta_{\mathcal{A}}(H_i^{Cn}, H_i^C)$ .

**Lemma 11.**  $\Delta_{\mathcal{A}}(H_i^{Cn}, H_i^C) \leq \mathbf{Adv}_{\text{cROR}_u}(p + 2q_2 + \sigma_2, \ell_2)$ .

*Proof.*  $H_i^{Cn}$  and  $H_i^C$  have different behavior only when  $i$ -th  $\text{ROR}$  call is  $\text{ROR}_u[\pi](len, n_{cpu})$  with  $S.G.\text{flag}_{n_{cpu}} = 0$ . So we assume that  $i$ -th  $\text{ROR}$  call is the case. This lemma can be proved similarly to the Lemma 10.

With Lemma 2, 5, 6, 7, 8, 9, 10, and 11, we can conclude the proof of Lemma 3.

#### 5.4 Proof of Lemma 4

**5.4.1 Proof of (4)** Instead of proving the security of the whole  $\text{pREF}_f$  game, we first prove prf security of COMP function given the initial state  $S.k_{\text{next}}$  has sufficient entropy, named COMP prf (CPRF) game described in Algorithm 10.

---

**Algorithm 10** Oracles for CPRF Game
 

---

**Procedure** INIT()  
 1:  $b \leftarrow_{\$} \{0, 1\}$   
 2:  $E \leftarrow_{\$} \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$   
 3:  $k \leftarrow_{\$} \{0, 1\}^n$

**Procedure** CPRF[ $E$ ]( $I$ )  
 1: **if**  $b = 0$  **then**  
 2:      $z \leftarrow_{\$} \{0, 1\}^n$   
 3: **else**  
 4:      $z \leftarrow \text{COMP}[E](0, \text{const}, 0^n \parallel k \parallel I)$   
 5: **return**  $z$

---

With oracles in Algorithm 10, CPRF game process is defined like below.  
**CPRF game.**

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $E, E^{-1}$  multiple time and  $\mathcal{A}_1$  queries CPRF[ $E$ ]( $I$ ) once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

In the CPRF game, we can bound the adversary's advantage using the following lemma.

**Lemma 12.** *Let  $\text{Adv}_{\text{CPRF}}(p, \ell)$  be the advantage upper bound against any adversary  $\mathcal{A}$  that makes at most  $p$  queries to  $E$  or  $E^{-1}$ , entropy input  $I$ 's length for CPRF is less than  $2n\ell$ . If  $p + \ell \leq 2^{n-1}$  and  $|\mathcal{W}| \leq 2^{0.5n}$ , the following inequality holds.*

$$\text{Adv}_{\text{CPRF}}(p, \ell) \leq \frac{3p}{2^n} + \frac{\ell}{2^n} + \frac{1}{2^{0.5n}}.$$

*Proof.* Assume that  $\mathcal{A}_2$  can obtain input  $I$  for CPRF query and  $k$  at the end of the game. Note that additional information never degrades adversarial advantage. Then the transcript  $\tau$  is defined as follows.

- The CPRF call input  $(I_1, \dots, I_\ell, \text{len})$  where

$$\begin{aligned} \text{len} &= |I|, \\ (I_1, \dots, I_{\ell-1}, I'_\ell) &\xleftarrow{2n} I, \end{aligned}$$

and  $I_\ell = 0^{2n-rem} \parallel I'_\ell$  where

$$rem = len - 2n(\lceil len/2n \rceil - 1).$$

- The CPRF call output  $z$ .
- Query results for block cipher  $E$ ,  $(k, x, y)$  and their set, subtranscript  $\tau_p$ .
- Additionally returned  $k$ .

For the notational convenience, let  $\tau_q$  be  $(I_0, I_1, \dots, I_\ell, z)$  where  $I_0 = 0^n \parallel k$ . Then, a transcript is the form of

$$\tau = (\tau_p, \tau_q).$$

And let  $p_i$  be the number of queries in  $\tau_p$  such that  $(\cdot, \cdot \parallel (0^{n/2} \parallel (i+1) \cdot 2n \parallel 0^{n/4}) \oplus IV, \cdot)$  for  $0 \leq i \leq \ell-1$ ,  $p_\ell$  be the number of queries in  $\tau_p$  such that  $(\cdot, \cdot \parallel (0^{n/2} \parallel 2n + len \parallel 0^{n/8} \parallel 1^{n/8}) \oplus IV, \cdot)$ , and  $T$  be the size of strong space for  $E$ . Note that  $T \geq 2^{2n} - 2^{0.5n}$ . We define a bad event like below.

- $\text{bad} : (I_0, \text{const} \parallel (0^{n/2} \parallel 2n \parallel 0^{n/4}) \oplus IV, \cdot) \in \tau_p$ .

Then, we can get the probability that a bad event happens in the ideal world.

$$\mathbf{p}_{\text{id}}[\text{bad}] = \frac{p_0}{2^n}.$$

For any good transcript  $\tau = (\tau_p, \tau_q)$ , we have

$$\begin{aligned} & \frac{\mathbf{p}_{\text{re}}[(\tau_p, \tau_q)]}{\mathbf{p}_{\text{id}}[(\tau_p, \tau_q)]} \\ &= \frac{\Pr[E \vdash \tau_p] \Pr[E \vdash \tau_q \mid h_0 \wedge E \vdash \tau_p]}{\Pr[E \vdash \tau_p] \Pr[z]} \\ &= \frac{\Pr[E \vdash \tau_q \mid E \vdash \tau_p]}{1/2^n} \end{aligned}$$

For  $0 \leq i \leq \ell$ , we recursively define a random variable  $H_i$  as

$$H_{i+1} = B[E](h_i, 2(i+1)n, I_i)$$

where  $H_0 = \text{const}$ . We define the following events.

- $\text{FRESH}_i : (\cdot, H_i \parallel (0^{n/2} \parallel (i+1) \cdot 2n \parallel 0^{n/4}) \oplus IV, \cdot) \notin \tau_p$  for  $1 \leq i \leq \ell-1$ .
- $\text{FRESH}_\ell : (\cdot, H_\ell \parallel (0^{n/2} \parallel len + 2n \parallel 0^{n/8} \parallel 0^{n/8}) \oplus IV, \cdot) \notin \tau_p$ .

Then, we have two following inequalities.

$$\begin{aligned} \Pr \left[ \text{FRESH}_i \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{i-1} \text{FRESH}_j \right) \right] &\geq 1 - \frac{p_i 2^n}{T - p - \ell - 1}, \\ \Pr \left[ B'[E](h_\ell, 2n + len, I_\ell) = z \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=0}^{\ell} \text{FRESH}_j \right) \right] \\ &\geq \frac{2^n - |\mathcal{W}| - p - \ell}{2^{2n}}. \end{aligned}$$



We can make a lower bound for good transcript probability with these two inequalities.

$$\begin{aligned}
 & \frac{\mathbf{pre}[(\tau_p, \tau_q)]}{\mathbf{pid}[(\tau_p, \tau_q)]} \\
 &= \frac{\Pr[E \vdash \tau_q \mid E \vdash \tau_p]}{1/2^n} \\
 &\geq 2^n \cdot \Pr \left[ \bigwedge_{j=1}^{\ell} \text{FRESH}_j \mid E \vdash \tau_p \right] \\
 &\quad \cdot \Pr \left[ E \vdash \tau_q \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{\ell} \text{FRESH}_j \right) \right] \\
 &\geq 2^n \cdot \left( \prod_{i=1}^{\ell} \Pr \left[ \text{FRESH}_i \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{i-1} \text{FRESH}_j \right) \right] \right) \\
 &\quad \cdot \Pr \left[ B'[E](h_\ell, 2n + \ell n, I_\ell) = z \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{\ell} \text{FRESH}_j \right) \right] \\
 &\geq \left( 1 - \sum_{i=1}^{\ell} \frac{p_i 2^n}{T - p - \ell - 1} \right) \left( 1 - \frac{2^{0.5n} + p + \ell}{2^n} \right) \\
 &\geq 1 - \left( \frac{3p}{2^n} + \frac{\ell}{2^n} + \frac{1}{2^{0.5n}} \right) \tag{13}
 \end{aligned}$$

From bad probability and good probability, we can conclude the proof.

And because  $|I_{cpu}| = n$ , we can show below inequality with triangle inequality.

$$\mathbf{Adv}_{\text{prf}}(p, \ell) \leq \mathbf{Adv}_{\text{CPRF}}(p, \ell) + 2\mathbf{Adv}_{\text{CPRF}}(p, 1).$$

With this inequality and Lemma 12, we can prove (4).

**5.4.2 Proof of (2)** Instead of proving the security of the whole M-EXT game, we first prove prf security of COMP function given the input to COMP has sufficient entropy, named multi-entropy (M-ENT) game described in Algorithm 11.

**Algorithm 11** Oracles for Multi-Entropy Security Game

---

**Procedure** INIT()

- 1:  $b \leftarrow_{\$} \{0, 1\}$
- 2:  $E \leftarrow_{\$} \Pi_w(2n, 2n, \mathcal{K}, \mathcal{W})$

**Procedure** M-ENT[ $E$ ]( $inc, h_1, I$ )

- 1: **if**  $b = 0$  **then**
- 2:    $s \leftarrow_{\$} \{0, 1\}^n$
- 3: **else**
- 4:    $s \leftarrow \text{COMP}[E](inc, h_1, I)$
- 5: **return** ( $inc, h_1, s$ )

---

With oracles in Algorithm 11, M-ENT game process is defined like below.  
**M-ENT game.**

1. Oracle runs INIT() procedure.
2. Adversary  $\mathcal{A}_2$  queries  $E, E^{-1}$  and  $\mathcal{A}_1$  queries M-ENT[ $E$ ]( $inc, h_1, I, len$ ) multiple time, and get returned value.
3. Adversary outputs  $b' \in \{0, 1\}$ , if  $b' = b$ , adversary wins.

The adversarial advantage for M-ENT game is defined as follows.

- $\text{Adv}_{\text{M-ENT}}(p, t, \sigma, \lambda)$  : The advantage upper bound against any  $\lambda$ -legitimate adversary  $\mathcal{A}$  that makes at most  $p$  queries to  $E$  or  $E^{-1}$ ,  $t$  queries to M-ENT, and the total length of entropy input  $I$  is less than  $2n\sigma$ .

In the M-ENT game, we can bound the adversarial advantage using the following lemma.

**Lemma 13.** *If  $p + \sigma \leq 2^{n-1}$ , the following inequality holds:*

$$\text{Adv}_{\text{M-ENT}}(p, q, \sigma, \lambda) \leq \frac{3pq + 2q\sigma + \sigma^2}{2^n} + \frac{q}{2^{0.5n}} + \frac{p^2}{2^n} + \frac{pq}{2^\lambda}.$$

*Proof.* Assume that  $\mathcal{A}_2$  can obtain inputs  $(inc, h_1, I)$  for all M-ENT queries at the end of the game. Note that additional information never degrades adversarial advantage. Then the transcript  $\tau$  is defined as follows.

- The M-ENT call results,  $Q_j = (inc^j, h_1^j, I_1^j, \dots, I_{\ell_j}^j, len^j, y^j)$  where

$$len^j = |I^j|,$$

$$(I_1^j, \dots, I_{\ell_j-1}^j, I_{\ell_j}^j) \xleftarrow{2n} I^j,$$

and  $I_{\ell_j}^j = 0^{2n-rem} \parallel I_{\ell_j}^j$  where

$$rem = len^j - 2n(\lceil len^j / 2n \rceil - 1).$$

and  $\ell_j$  is the block length for  $j$ -th entropy input  $I^j$ . We denote  $\tau_q = (Q_1, \dots, Q_q)$ .

- query result for block cipher  $E$ ,  $(k, x, y)$  and their set, subtranscript  $\tau_p$ .

The transcript is form of  $\tau = (\tau_p, \tau_q)$ . Let  $p_u$  be the number of queries in  $\tau_p$  s.t.  $(\cdot, \cdot \parallel (0^{n/2} \parallel u \parallel 0^{n/4}) \oplus IV, \cdot)$ . It is obvious that  $\sum_{u=0}^{2^n-1} p_u \leq p$ . Let  $T$  be the size of strong space for  $E$ . Note that  $T \geq 2^{2n} - 2^{0.5n}$ . For each  $Q_j$ , define  $\ell'_j$  like below.

- The maximum number that satisfies  $(I_i^j, h_i^j \parallel (2^{n/2} \parallel inc^j + i \cdot 2n \parallel 0^{n/4}) \oplus IV, y_i^j) \in \tau_p$  for all  $1 \leq i \leq \ell'_j - 1$ , where  $h_{i+1}^j = \text{TRSum}(y_i^j) \oplus h_i^j$ .
- If  $\ell'_j = \ell_j - 1$  and  $(I_\ell^j, h_\ell^j \parallel (2^{n/2} \parallel inc^j + len^j \parallel 0^{n/8} \parallel 1^{n/8}) \oplus IV, \cdot) \in \tau_p$  where  $h_\ell^j = \text{TRSum}(y_{\ell-1}^j) \oplus h_{\ell-1}^j$ , we let  $\ell'_j = \ell_j$ .

Then we define bad events for M-ENT game like below.

- $\text{bad}_1$  : there exist two distinct  $(k, x \parallel \cdot, y), (k', x' \parallel \cdot, y') \in \tau_p$  such that  $x \oplus \text{TRSum}(y) = x' \oplus \text{TRSum}(y')$ .
- $\text{bad}_2^j$  : for all  $1 \leq i \leq \ell_j - 1$ ,  $(I_i^j, h_i^j \parallel (2^{n/2} \parallel inc^j + i \cdot 2n \parallel 0^{n/4}) \oplus IV, y_i^j) \in \tau_p$  and  $(I_\ell^j, h_\ell^j \parallel (2^{n/2} \parallel inc^j + len^j \parallel 0^{n/8} \parallel 1^{n/8}) \oplus IV, \cdot) \in \tau_p$  where  $h_{i+1}^j = \text{TRSum}(y_i^j) \oplus h_i^j$  for  $1 \leq i \leq \ell_j - 1$ . Namely,  $\ell'_j = \ell_j$ .

Then we can bound  $\text{bad}_1$  probability like below.

$$\mathbf{p}_{\text{id}}[\text{bad}_1] \leq \frac{p^2}{2} \frac{2^n}{T-1} \leq \frac{p^2}{2^n}.$$

To bound  $\mathbf{p}_{\text{id}}[\text{bad}_2^j \wedge \neg \text{bad}_1]$ , define a **potential chain** as the tuple  $(inc^j, h_1^j, \dots, h_\ell^j, t^j)$  for some  $\ell$ , which satisfies for all  $1 \leq i \leq \ell - 1$ ,  $(\cdot, h_i^j \parallel 2^{n/2} \parallel inc^j + i \cdot 2n \parallel 0^{n/4}, y_i^j) \in \tau_p$  and  $(\cdot, h_\ell^j \parallel 2^{n/2} \parallel inc^j + (\ell - 1)2n + t^j \parallel 0^{n/8} \parallel 1^{n/8}, \cdot) \in \tau_p$  where  $h_{i+1}^j = \text{TRSum}(y_i^j) \oplus h_i^j$  for  $1 \leq i \leq \ell - 1$  and  $1 < t^j \leq 2n$ . Then assuming  $\text{bad}_1$  doesn't happen, which means there is no collision in  $\tau_p$ , we can bound the number of potential chains as at most  $p$ . Therefore by the legitimacy of  $\mathcal{A}$ , for any  $\tau_j = (\tau_p, \tau_q \setminus Q_j)$  that satisfies  $\tau_p$  doesn't make  $\text{bad}_1$  event and  $\mathcal{I}^j$ , the random variable about  $I^j$ , we have

$$\mathbf{p}_{\text{id}}[\text{bad}_2^j | \tau_j] \leq p \cdot \text{Pred}(\mathcal{I}^j | \tau_j).$$

Let  $\Gamma_j$  be the set of all possible  $\tau_j$  and  $g\Gamma_j$  be the set of all possible  $\tau_j = (\tau_p, \tau_q \setminus Q_j)$  that satisfies  $\tau_p$  doesn't make  $\text{bad}_1$  event. Then with inequality above, we can get below inequality.

$$\begin{aligned} \mathbf{p}_{\text{id}}[\text{bad}_2^j \wedge \neg \text{bad}_1] &= \sum_{\tau_j \in \Gamma_j} \mathbf{p}_{\text{id}}[\tau_j] \cdot \mathbf{p}_{\text{id}}[\text{bad}_2^j \wedge \neg \text{bad}_1 | \tau_j] \\ &= \sum_{\tau_j \in g\Gamma_j} \mathbf{p}_{\text{id}}[\tau_j] \cdot \mathbf{p}_{\text{id}}[\text{bad}_2^j | \tau_j] \\ &\leq \sum_{\tau_j \in g\Gamma_j} \mathbf{p}_{\text{id}}[\tau_j] p \cdot \text{Pred}(\mathcal{I}^j | \tau_j) \\ &\leq p \cdot \text{Pred}(\mathcal{I}^j | \mathcal{T}_j) \leq \frac{p}{2^\lambda} \end{aligned}$$

where  $\mathcal{T}_j$  is random variable about  $\tau_j$ .

Therefore we can bound **bad** event probability.

$$\begin{aligned}
\mathbf{p}_{\text{id}}[\text{bad}] &= \mathbf{p}_{\text{id}} \left[ \text{bad}_1 \vee \left( \bigvee_{j=1}^q \text{bad}_2^j \right) \right] \\
&\leq \mathbf{p}_{\text{id}}[\text{bad}_1] + \sum_{j=1}^q \mathbf{p}_{\text{id}}[\text{bad}_2^j \wedge \neg \text{bad}_1] \\
&\leq \frac{p^2}{2^n} + \frac{pq}{2^\lambda}
\end{aligned} \tag{14}$$

For good transcript  $\tau = (\tau_p, \tau_q)$ , we have

$$\begin{aligned}
&\frac{\mathbf{p}_{\text{re}}[(\tau_p, \tau_q)]}{\mathbf{p}_{\text{id}}[(\tau_p, \tau_q)]} \\
&= \frac{\Pr[E \vdash \tau_p] \Pr[E \vdash \tau_q \mid E \vdash \tau_p]}{\Pr[E \vdash \tau_p] \Pr[y^1, \dots, y^q]} \\
&= \frac{\Pr[E \vdash \tau_q \mid E \vdash \tau_p]}{1/2^{qn}} \\
&= 2^{qn} \Pr[E \vdash \tau_q \mid E \vdash \tau_p].
\end{aligned}$$

For  $1 \leq j \leq q$ ,  $\ell'_t < i \leq \ell_t$ , and any  $v_i^j \in \{0, 1\}^{2n}$ , let  $V_j$  be a tuple of intermediate values  $(v_{\ell'_t+1}^1, \dots, v_{\ell'_t+1}^1, v_{\ell'_t+1}^2, \dots, v_{\ell'_t+1}^j)$  that satisfies

$$y^t = \text{TRSum}(v_{\ell'_t}^t) \oplus h_{\ell'_t}^t,$$

where  $h_{i+1}^t = \text{TRSum}(v_i^t) \oplus h_i^t$ , with  $h_{\ell'_t+1}^t = \text{TRSum}(y_{\ell'_t}^t) \oplus h_{\ell'_t}^t$ , for  $1 \leq t \leq j$ .

For each  $V_j$ , we define a set  $\tau_p^j$  that contains:

- For  $1 \leq t \leq j$  and  $\ell'_t < i \leq \ell_t - 1$ , the tuple  $(I_i^t, h_i^t \parallel (0^{n/2} \parallel \text{inc}^t + i \cdot 2n \parallel 0^{n/4}) \oplus IV, v_i^t)$ .
- For  $1 \leq t \leq j$ , the tuple  $(I_{\ell'_t}^t, h_{\ell'_t}^t \parallel (0^{n/2} \parallel \text{inc}^t + \text{len}^t \parallel 0^{n/8} \parallel 1^{n/8}) \oplus IV, v_{\ell'_t}^t)$ .

For the rest of the proof, we assume  $\tau_p^i$  as a pseudo-transcript, which means  $E \vdash \tau_p^i$  means for all  $(k, x, y) \in \tau_p^i$ ,  $E_k(x) = y$ .

And let  $\Gamma_j'$  be the set of all possible  $\tau_p^j$  that satisfies the following condition:

- For any  $g \in \{0, 1\}^{n/4}$  and every distinct  $(k, h \parallel (0^{n/2} \parallel g \parallel 0) \oplus IV, y), (k', h' \parallel (0^{n/2} \parallel g \parallel 0) \oplus IV, y') \in \tau_p \cup \tau_p^j$ ,  $h \oplus \text{TRSum}(y) \neq h' \oplus \text{TRSum}(y')$ .

By the definition of  $\Gamma_j'$ , we can acknowledge that for any  $\tau_p^j \in \Gamma_j'$ ,  $E \vdash \tau_p \cup \tau_p^j$  also means  $E \vdash (\tau_p, Q_1, \dots, Q_j)$ . Let  $E \vdash \Gamma_j'$  be the event that there exists  $\tau \in \Gamma_j'$  that satisfies  $E \vdash \tau$  with some abuse of notation. Then with  $\Gamma_0' = \emptyset$ , we have

$$\begin{aligned}
& \Pr[E \vdash \tau_q \mid E \vdash \tau_p] \\
& \geq \Pr[E \vdash \Gamma'_q \mid E \vdash \tau_p] \\
& \geq \prod_{j=1}^q \Pr[E \vdash \Gamma'_j \mid E \vdash \tau_p \wedge E \vdash \Gamma'_{j-1}].
\end{aligned}$$

For  $1 \leq j \leq q$ ,  $\ell'_j < i \leq \ell_j$ , we recursively define a random variable  $H_i^j$  as

$$H_{i+1}^j = B[E](H_i^j, inc^j + i \cdot 2n, I_i^j).$$

Where  $H_{\ell'_j}^j = h_{\ell'_j}^j$ . With any  $\tau_p^{j-1} \in \Gamma'_{j-1}$ , define following events.

- $\text{FRESH}_i^j : (\cdot, H_i^j \parallel (0^{n/2} \parallel inc^j + i \cdot 2n \parallel 0^{n/4}) \oplus IV, \cdot) \notin \tau_p \cup \tau_p^{j-1}$  for  $\ell'_j + 1 \leq i \leq \ell_j - 1$ .
- $\text{FRESH}_{\ell_j}^j : (\cdot, H_{\ell_j}^j \parallel (0^{n/2} \parallel inc^j + len^j \parallel 0^{n/8} \parallel 0^{n/8}) \oplus IV, \cdot) \notin \tau_p \cup \tau_p^{j-1}$ .

Then we have two below inequality.

- For  $\ell'_j + 1 \leq i \leq \ell_j$ ,

$$\Pr \left[ \text{FRESH}_i^j \mid E \vdash \tau_p \cup \tau_p^{j-1} \wedge \left( \bigwedge_{t=\ell'_j+1}^{i-1} \text{FRESH}_t^j \right) \right] \geq 1 - \frac{(p_{inc^j+i} + j - 1)2^n}{T - p - \sigma + 1}.$$

–

$$\begin{aligned}
& \Pr \left[ B'[E](h_{\ell'_j}^j, inc^j + len^j, I_{\ell'_j}^j) = s_j \mid E \vdash \tau_p \cup \tau_p^{j-1} \wedge \left( \bigwedge_{t=\ell'_j+1}^{\ell_j} \text{FRESH}_t^j \right) \right] \\
& \geq \frac{2^n - |W| - p - \sigma}{2^{2n}}.
\end{aligned}$$

Similarly to (13), we have

$$\begin{aligned}
& \Pr[E \vdash \Gamma'_j \mid E \vdash \tau_p \wedge E \vdash \tau_p^{j-1}] \\
& \geq \left( 1 - \left( \frac{2p + 2q\ell_j}{2^n} + \frac{2^{0.5n} + p + \sigma}{2^n} \right) \right) \cdot 1/2^n.
\end{aligned}$$

for all  $\tau_p^{j-1} \in \Gamma'_{j-1}$ , then it also means

$$\begin{aligned}
& \Pr[E \vdash \Gamma'_j \mid E \vdash \tau_p \wedge E \vdash \Gamma'_{j-1}] \\
& \geq \left( 1 - \left( \frac{2p + 2q\ell_j}{2^n} + \frac{2^{0.5n} + p + \sigma}{2^n} \right) \right) \cdot 1/2^n. \tag{15}
\end{aligned}$$

With (15), we can bound good transcript probability. With that bound and (14), we can conclude the proof.

$$\begin{aligned}
& \frac{\mathbf{pre}[(\tau_p, \tau_q)]}{\mathbf{pid}[(\tau_p, \tau_q)]} \\
& \geq 2^{qn} \prod_{j=1}^q \Pr[E \vdash \Gamma'_j | E \vdash \tau_p \wedge E \vdash \Gamma'_{j-1}] \\
& \geq 2^{qn} \prod_{j=1}^q \left( \left( 1 - \left( \frac{2p + 2q\ell_j}{2^n} + \frac{2^{0.5n} + p + \sigma}{2^n} \right) \right) \cdot 1/2^n \right) \\
& \geq 1 - \left( \frac{3pq + 2q\sigma + \sigma^2}{2^n} + \frac{q}{2^{0.5n}} \right).
\end{aligned}$$

We can show the below inequality with triangle inequality.

$$\mathbf{Adv}_{\text{M-EXT}}(p, q, \sigma) \leq \mathbf{Adv}_{\text{M-ENT}}(p, q, \sigma) + 2q\mathbf{Adv}_{\text{CPRF}}(p, \sigma).$$

With this inequality and because M-EXT security means the indistinguishability of 'base key' and 'crng key', we can trivially prove (2) with M-ENT and prf security of blake2s with Lemma 12 and Lemma 13.

**5.4.3 Proof of (3)** Assume that  $\mathcal{A}_2$  can obtain input  $I$  for  $\text{pREF}_a$  query and  $k$  at the end of the game. Note that additional information never degrades adversarial advantage. Then the transcript  $\tau$  is defined as follows.

- The  $\text{pREF}_a$  call input  $(I_1, \dots, I_\ell, \text{len})$  where

$$\begin{aligned}
& \text{len} = |I|, \\
& (I_1, \dots, I_{\ell-1}, I_\ell) \xleftarrow{2n} I,
\end{aligned}$$

- The RPRF call output  $h$ .
- Query results for block cipher  $E$ ,  $(k, x, y)$  and their set, subtranscript  $\tau_p$ .
- Additionally returned  $k$ .

For the notational convenience, let  $\tau_q$  be  $(I_0, I_1, \dots, I_\ell, h)$  where  $I_0 = 0^n \parallel k$ . Then, a transcript is the form of

$$\tau = (\tau_p, \tau_q).$$

And let  $p_i$  be the number of queries in  $\tau_p$  such that  $(\cdot, \cdot \parallel (0^{n/2} \parallel (i+1) \cdot 2n \parallel 0^{n/4}) \oplus IV, \cdot)$  for  $0 \leq i \leq \ell$ , and  $T$  be the size of strong space for  $E$ . Note that  $T \geq 2^{2n} - 2^{0.5n}$ . We define a bad event like below.

- **bad** :  $(I_0, \text{const} \parallel (0^{n/2} \parallel 2n \parallel 0^{n/4}) \oplus IV, \cdot) \in \tau_p$ .

Then, we can get the probability that a bad event happens in the ideal world.

$$\mathbf{p}_{\text{id}}[\text{bad}] = \frac{p_0}{2^n}.$$

For any good transcript  $\tau = (\tau_p, \tau_q)$ , we have

$$\begin{aligned} & \frac{\mathbf{p}_{\text{re}}[(\tau_p, \tau_q)]}{\mathbf{p}_{\text{id}}[(\tau_p, \tau_q)]} \\ &= \frac{\Pr[E \vdash \tau_p] \Pr[E \vdash \tau_q \mid h_0 \wedge E \vdash \tau_p]}{\Pr[E \vdash \tau_p] \Pr[y]} \\ &= \frac{\Pr[E \vdash \tau_q \mid E \vdash \tau_p]}{1/2^n} \end{aligned}$$

For  $0 \leq i \leq \ell$ , we recursively define a random variable  $H_i$  as

$$H_{i+1} = B[E](h_i, 2(i+1)n, I_i)$$

where  $H_0 = \text{const}$ . We define the following events.

$$- \text{FRESH}_i : (\cdot, H_i \parallel (0^{n/2} \parallel (i+1) \cdot 2n \parallel 0^{n/4}) \oplus IV, \cdot) \notin \tau_p \text{ for } 1 \leq i \leq \ell.$$

Then, we have two following inequalities.

$$\begin{aligned} \Pr \left[ \text{FRESH}_i \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{i-1} \text{FRESH}_j \right) \right] &\geq 1 - \frac{p_i 2^n}{T - p - \ell + 1}, \\ \Pr \left[ B[E](h_\ell, 2n + \ell n, I_\ell) = h \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=0}^{\ell} \text{FRESH}_j \right) \right] \\ &\geq \frac{2^n - |\mathcal{W}| - p - \ell}{2^{2n}}. \end{aligned}$$

We can make a lower bound for good transcript probability with these two inequalities.

$$\begin{aligned}
& \frac{\Pr[(\tau_p, \tau_q)]}{\Pr[(\tau_p, \tau_q)]} \\
&= \frac{\Pr[E \vdash \tau_q \mid E \vdash \tau_p]}{1/2^n} \\
&\geq 2^n \cdot \Pr \left[ \bigwedge_{j=1}^{\ell} \text{FRESH}_j \mid E \vdash \tau_q \right] \\
&\quad \cdot \Pr \left[ E \vdash \tau_q \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{\ell} \text{FRESH}_j \right) \right] \\
&\geq 2^n \cdot \left( \prod_{i=1}^{\ell} \Pr \left[ \text{FRESH}_i \mid E \vdash \tau_p \wedge \left( \bigwedge_{j=1}^{i-1} \text{FRESH}_j \right) \right] \right) \\
&\quad \cdot \Pr \left[ B[E](h_{\ell}, 2n + \ell n, I_{\ell}) = h \mid E \vdash \tau_p \wedge \left( \bigwedge_{i=1}^{\ell} \text{FRESH}_i \right) \right] \\
&\geq \left( 1 - \sum_{i=1}^{\ell} \frac{p_i 2^n}{T - p - \ell - 1} \right) \left( 1 - \frac{2^{0.5n} + p + \ell}{2^n} \right) \\
&\geq 1 - \left( \frac{3p}{2^n} + \frac{\ell}{2^n} + \frac{1}{2^{0.5n}} \right) \tag{16}
\end{aligned}$$

From bad probability and good probability, we can conclude the proof.

#### 5.4.4 Proof of (5) Define a transcript $\tau$ as

$$\tau = \left( k_{\text{base}}^*, \{(X_i, Y_i)\}_{i=1}^{\ell}, \{(x_j, y_j)\}_{j=1}^p \right)$$

where elements of the  $\tau$  is defined as follows:

- $k_{\text{base}}^*$ : the random key picked in the security game, the key is given to the attacker when the attacker finishes querying,
- $\{(X_i, Y_i)\}_{i=1}^{\ell}$ : for every  $i = 1, \dots, \ell$ ,
  - In real world,  $Y_i = X_i +_{n/8} \pi(X_i)$
  - In ideal world,  $Y_i \leftarrow_{\$} \{0, 1\}^{2n}$ ,
 where  $X_1 = Z \parallel k_{\text{base}}^* \parallel 0$ , and  $X_i$  has  $Z \parallel Y_1[n:] \parallel (i-2)$  format for  $i = 2, \dots, \ell$ .  
 $Y_1[n:]$  is given to the attacker when the attacker finishes querying,
- $\{(x_j, y_j)\}_{j=1}^p$ : for every  $j = 1, \dots, p$   $y_j = \pi(x_j)$ .

Now we define bad cases:

- $\text{bad}_1 \Leftrightarrow k_{\text{base}}^* = Y_1[n:]$ .



- $\text{bad}_2 \Leftrightarrow \exists i \neq j \in [\ell] \text{ s.t. } Y_i -_{n/8} X_i = Y_j -_{n/8} X_j.$
- $\text{bad}_3 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = \mathbf{k}_{\text{base}}^*.$
- $\text{bad}_4 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = Y_1[n:].$
- $\text{bad}_5 \Leftrightarrow \exists i \in [\ell] \text{ and } j \in [p] \text{ s.t. } Y_i = X_i +_{n/8} y_j.$

Note that we only consider the ideal world when analyzing bad cases. For  $\text{bad}_1$ , both keys are uniformly randomly picked from  $\{0, 1\}^n$ . Hence

$$\Pr[\text{bad}_1] \leq \frac{1}{2^n}.$$

For  $\text{bad}_2$ ,  $Y_i$  and  $Y_j$  are uniform randomly picked from  $\{0, 1\}^{2n}$  and they are independent with  $X_i$  and  $X_j$ . Hence

$$\Pr[\text{bad}_2] \leq \binom{\ell}{2} \frac{1}{2^{2n}} \leq \frac{\ell^2}{2^{2n}}.$$

For  $\text{bad}_3$  and  $\text{bad}_4$ , as  $\mathbf{k}_{\text{base}}^*$  and  $Y_1[n:]$  is chosen uniform randomly from  $\{0, 1\}^n$ ,

$$\begin{aligned} \Pr[\text{bad}_3] &\leq \frac{p}{2^n}, \\ \Pr[\text{bad}_4] &\leq \frac{p}{2^n}. \end{aligned}$$

For  $\text{bad}_5$ , in ideal world,  $Y_i$  is chosen uniform randomly from  $\{0, 1\}^{2n}$  independent with  $X_i$ .

$$\Pr[\text{bad}_5] \leq \frac{\ell p}{2^{2n}}.$$

Therefore we have

$$\Pr[\text{bad}] \leq \frac{p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

For good case,

$$\frac{\Pr[\tau]}{\Pr_{\text{id}}[\tau]} = \frac{1}{\left(\frac{1}{2^{2n}}\right)^{\ell+p}} \geq 1.$$

Therefore, by H-Coefficient technique,

$$\mathbf{Adv}_{\text{bROR}_k}(p, \ell) \leq \frac{1 + 2p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

### 5.5 Proof of (6)

Define a transcript  $\tau$  as

$$\tau = \left( \text{c.key}^*, \{(X_i, Y_i)\}_{i=1}^{\ell}, \{(x_j, y_j)\}_{j=1}^p, r \right)$$

where elements of the  $\tau$  is defined as follows:

- $\text{c\_key}^*$ : the random key picked in the security game, the key is given to the attacker when the attacker finishes querying,
- $\{(X_i, Y_i)\}_{i=1}^\ell$ : for every  $i = 1, \dots, \ell$ 
  - In real world,  $Y_i = X_i +_{n/8} \pi(X_i)$
  - In ideal world,  $Y_i \leftarrow_{\$} \{0, 1\}^{2n}$

where

$$X_1 = Z \parallel \text{c\_key}^* \parallel 0,$$

and  $X_i$  has  $Z \parallel \text{c\_key}^* \parallel (i-1)$  format for  $i = 2, \dots, \ell$ ,

- $\{(x_j, y_j)\}_{j=1}^p$ : for every  $j = 1, \dots, p$   $y_j = \pi(x_j)$ .

Now we define bad cases:

- $\text{bad}_1 \Leftrightarrow \exists i \neq j \in [\ell] \text{ s.t. } Y_i -_{n/8} X_i = Y_j -_{n/8} X_j$ .
- $\text{bad}_2 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = \text{c\_key}^*$ .
- $\text{bad}_3 \Leftrightarrow \exists i \in [\ell] \text{ and } j \in [p] \text{ s.t. } Y_i = X_i +_{n/8} y_j$ .

For  $\text{bad}_1$ ,  $Y_i$  and  $Y_j$  are uniform randomly picked from  $\{0, 1\}^{2n}$  and they are independent with  $X_i$  and  $X_j$ . Hence

$$\Pr[\text{bad}_1] \leq \frac{\ell^2}{2^{2n}}.$$

For  $\text{bad}_2$ , as  $\text{c\_key}^*$  is chosen uniform randomly from  $\{0, 1\}^n$ ,

$$\Pr[\text{bad}_2] \leq \frac{p}{2^n}.$$

For  $\text{bad}_3$ , in ideal world,  $Y_i$  is chosen uniform randomly from  $\{0, 1\}^{2n}$  independent with  $X_i$ .

$$\Pr[\text{bad}_3] \leq \frac{\ell p}{2^{2n}}.$$

Therefore we have

$$\Pr[\text{bad}] \leq \frac{p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

For good case,

$$\frac{\text{pre}[\tau]}{\text{pid}[\tau]} = \frac{\frac{1}{(2^{2n})^{\ell+p}}}{\left(\frac{1}{2^{2n}}\right)^{\ell+p}} \geq 1.$$

Therefore, by H-Coefficient technique,

$$\mathbf{Adv}_{\text{cROR}_k}(p, \ell) \leq \frac{p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

### 5.6 Proof of (7)

Define a transcript  $\tau$  as

$$\tau = \left( \mathbf{k}_{\text{base}}^*, \{(X_i, Y_i)\}_{i=1}^\ell, \{(x_j, y_j)\}_{j=1}^p \right)$$

where elements of the  $\tau$  is defined as follows:

- $\mathbf{k}_{\text{base}}^*$ : the random key picked in the security game, the key is given to the attacker when the attacker finishes querying,
  - $\{(X_i, Y_i)\}_{i=1}^\ell$ : for every  $i = 1, \dots, \ell$ 
    - In real world,  $Y_i = X_i +_{n/8} \pi(X_i)$
    - In ideal world,  $Y_i \leftarrow_{\$} \{0, 1\}^{2n}$
- where  $X_1 = Z \parallel \mathbf{k}_{\text{base}}^* \parallel 0$ ,  $X_2 = Z \parallel Y_1[n:] \parallel 0$ , and  $X_i$  has  $Z \parallel Y_2[n:] \parallel (i-2)$  format for  $i = 3, \dots, \ell$ .  $Y_1[n:]$  and  $Y_2[n:]$  are given to the attacker when the attacker finishes querying,
- $\{(x_j, y_j)\}_{j=1}^p$ : for every  $j = 1, \dots, p$   $y_j = \pi(x_j)$ .

Now we define bad cases:

- $\text{bad}_1 \Leftrightarrow \mathbf{k}_{\text{base}}^* = Y_1[n:]$ .
- $\text{bad}_2 \Leftrightarrow \exists i \neq j \in [\ell] \text{ s.t. } Y_i -_{n/8} X_i = Y_j -_{n/8} X_j$ .
- $\text{bad}_3 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = \mathbf{k}_{\text{base}}^*$ .
- $\text{bad}_4 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = Y_1[n:]$ .
- $\text{bad}_5 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = Y_2[n:]$ .
- $\text{bad}_6 \Leftrightarrow \exists i \in [\ell] \text{ and } j \in [p] \text{ s.t. } Y_i = X_i +_{n/8} y_j$ .

Note that we only consider the ideal world when analyzing bad cases. For  $\text{bad}_1$ , both keys are uniformly randomly picked from  $\{0, 1\}^n$ . Hence

$$\Pr[\text{bad}_1] \leq \frac{1}{2^n}.$$

For  $\text{bad}_2$ ,  $Y_i$  and  $Y_j$  are uniform randomly picked from  $\{0, 1\}^{2n}$  and they are independent with  $X_i$  and  $X_j$ . Hence

$$\Pr[\text{bad}_2] \leq \binom{\ell}{2} \frac{1}{2^{2n}} \leq \frac{\ell^2}{2^{2n}}.$$

For  $\text{bad}_3, \text{bad}_4$  and  $\text{bad}_5$ , as  $\mathbf{k}_{\text{base}}^*, Y_1[n:]$  and  $Y_2[n:]$  is chosen uniform randomly from  $\{0, 1\}^n$ ,

$$\Pr[\text{bad}_3] \leq \frac{p}{2^n},$$

$$\Pr[\text{bad}_4] \leq \frac{p}{2^n},$$

$$\Pr[\text{bad}_5] \leq \frac{p}{2^n}.$$

For  $\text{bad}_6$ , in ideal world,  $Y_i$  is chosen uniform randomly from  $\{0, 1\}^{2n}$  independent with  $X_i$ .

$$\Pr[\text{bad}_6] \leq \frac{\ell p}{2^{2n}}.$$

Therefore we have

$$\Pr[\text{bad}] \leq \frac{1 + 3p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

For good case,

$$\frac{\text{pre}[\tau]}{\text{pid}[\tau]} = \frac{\frac{1}{(2^{2n})^{\ell+p}}}{\left(\frac{1}{2^{2n}}\right)^{\ell+p}} \geq 1.$$

Therefore, by H-Coefficient technique,

$$\text{Adv}_{\text{bROR-u}}(p, \ell) \leq \frac{1 + 3p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

### 5.7 Proof of (8)

Define a transcript  $\tau$  as

$$\tau = \left( \text{c\_key}^*, \{(X_i, Y_i)\}_{i=1}^{\ell}, \{(x_j, y_j)\}_{j=1}^p \right)$$

where elements of the  $\tau$  is defined as follows:

- $\text{c\_key}^*$ : the random key picked in the security game, the key is given to the attacker when the attacker finishes querying,
- $\{(X_i, Y_i)\}_{i=1}^{\ell}$ : for every  $i = 1, \dots, \ell$ 
  - In real world,  $Y_i = X_i +_{n/8} \pi(X_i)$
  - In ideal world,  $Y_i \leftarrow_{\$} \{0, 1\}^{2n}$

where

$$X_1 = Z \parallel \text{c\_key}^* \parallel 0,$$

and  $X_i$  has  $Z \parallel Y_1[n:] \parallel (i-1)$  format for  $i = 2, \dots, \ell$ .  $Y_1[n:]$  is given to the attacker when the attacker finishes querying,

- $\{(x_j, y_j)\}_{j=1}^p$ : for every  $j = 1, \dots, p$   $y_j = \pi(x_j)$ .

Now we define bad cases:

- $\text{bad}_1 \Leftrightarrow \exists i \neq j \in [\ell] \text{ s.t. } Y_i -_{n/8} X_i = Y_j -_{n/8} X_j$ .
- $\text{bad}_2 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = \text{c\_key}^*$ .
- $\text{bad}_3 \Leftrightarrow \exists j \in [p] \text{ s.t. } x_j[n/2 : 3n/2 - 1] = Y_1[n:]$ .
- $\text{bad}_4 \Leftrightarrow \exists i \in [\ell] \text{ and } j \in [p] \text{ s.t. } Y_i = X_i +_{n/8} y_j$ .

For  $\text{bad}_1$ ,  $Y_i$  and  $Y_j$  are uniform randomly picked from  $\{0, 1\}^{2n}$  and they are independent with  $X_i$  and  $X_j$ . Hence

$$\Pr[\text{bad}_1] \leq \frac{\ell^2}{2^{2n}}.$$

For  $\text{bad}_2$  and  $\text{bad}_3$ , as  $\text{c\_key}^*$  and  $Y_1[n:]$  are chosen uniform randomly from  $\{0, 1\}^n$ ,

$$\Pr[\text{bad}_2] \leq \frac{p}{2^n},$$

$$\Pr[\text{bad}_3] \leq \frac{p}{2^n}.$$

For  $\text{bad}_4$ , in ideal world,  $Y_i$  is chosen uniform randomly from  $\{0, 1\}^{2n}$  independent with  $X_i$ .

$$\Pr[\text{bad}_4] \leq \frac{\ell p}{2^{2n}}.$$

Therefore we have

$$\Pr[\text{bad}] \leq \frac{2p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

For good case,

$$\frac{\text{pre}[\tau]}{\text{id}[\tau]} = \frac{\frac{1}{(2^{2n})^{\ell+p}}}{\left(\frac{1}{2^{2n}}\right)^{\ell+p}} \geq 1.$$

Therefore, by H-Coefficient technique,

$$\mathbf{Adv}_{\text{cROR}_{\text{u}}}(p, \ell) \leq \frac{2p}{2^n} + \frac{\ell^2 + \ell p}{2^{2n}}.$$

## 6 Tight attack for Linux-DRBG.

In this section, we briefly explain attacks to demonstrate the tightness of our proof. We will present two attacks: the first attack can be executed when  $\lambda < n$  with  $O(2^{\lambda/2})$  complexity, and the second attack has  $O(2^{n/2})$  complexity.

**ATTACK 1:** When  $\lambda < n$ , a  $\lambda$ -legitimate adversary  $\mathcal{A}$  can win the robustness game with high probability with the following method.

1. Make  $\mathcal{A}_1$  to pick entropy inputs uniformly random from set  $\mathcal{T}$  where  $|\mathcal{T}| = 2^\lambda$ , regardless of query result. Note that  $\mathcal{A}$  is still  $\lambda$ -legitimate.
2. For any  $S^* \in \mathcal{S}$ , and distinct  $I_1 \cdots, I_p \in \mathcal{T}$ ,  $\mathcal{A}_2$  simulates  $S_i \leftarrow \text{refresh}_f[E](S^*, I_i, 0^n)$  by repeatedly querying  $E$  and calculate  $p$   $S_i$  values.

3. For any positive integer  $c$ ,  $\mathcal{A}_2$  simulates  $\text{next}_k[\pi](S_i, 3n, 1)$  by repeatedly querying  $\pi$  and calculates  $p$  output random bits. Then save the outputs in  $\mathcal{X}$ .
4.  $\mathcal{A}_2$  queries  $\text{SET}(S^*)$  and  $\mathcal{A}_1$  picks  $I$  and queries  $\text{REF}_f[E](I, \lambda, 0^n)$ , then  $\mathcal{A}_2$  makes  $\text{ROR}_u[\pi](3n, 1)$  to get random bits. Repeat this procedure  $q$  times and save the values in  $\mathcal{Y}$ .
5. If  $\mathcal{X} \cap \mathcal{Y} = \emptyset$ ,  $\mathcal{A}_2$  outputs 0. Else,  $\mathcal{A}_2$  outputs 1.

To make an intersection, in the real world, it is sufficient to make the collision between entropy input and simulated entropy input. However, in the ideal world, the output bits are generated uniformly randomly. Therefore we have

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 0] = \frac{pq}{2^{3n}}$$

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 1] = 1 - \left(1 - \frac{p}{2^\lambda}\right)^q \geq \frac{pq}{2^\lambda} - \frac{(pq)^2}{2^{2\lambda+1}}.$$

Therefore, if  $p = q = 2^{\lambda/2}$ , the advantage of  $\mathcal{A}$  is sufficiently non-negligible.

**ATTACK 2:** A  $\lambda$ -legitimate adversary  $\mathcal{A}$  can win a robustness game with high probability with the following method.

1.  $\mathcal{A}$  picks key  $K \leftarrow_{\$} \{0, 1\}^n$  and  $\mathcal{A}$  simulates  $\text{next}_k[\pi](S, 3n, 1)$  as if  $S.\text{ckey}_1 = K$  and  $S.\text{G\_flag}_1 = 0$  by repeatedly querying  $\pi$ . Then save the outputs in  $\mathcal{X}$ . Repeat this procedure  $p$  times.
2.  $\mathcal{A}_1$  generates  $I$  with min-entropy  $\lambda$ , then queries  $\text{REF}_f[E](I, \lambda, 0^n)$  and  $\mathcal{A}_2$  queries  $\text{ROR}_k[\pi](3n, 1)$  and save the outputs in  $\mathcal{Y}$ . Repeat this procedure until  $|\mathcal{Y}|$  becomes  $q$ .
3. If  $\mathcal{X} \cap \mathcal{Y} = \emptyset$ ,  $\mathcal{A}$  outputs 0. Else,  $\mathcal{A}$  outputs 1.

To make an intersection, in the real world, it is sufficient to make the collision on  $S.\text{ckey}_1$ . However, in the ideal world, the output bits are generated uniformly randomly. Therefore we have

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 0] = \frac{pq}{2^{3n}}$$

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 1] = 1 - \left(1 - \frac{q}{2^n}\right)^p \geq \frac{pq}{2^n} - \frac{(pq)^2}{2^{2n+1}}.$$

Therefore, if  $p = q = 2^{n/2}$ , the advantage of  $\mathcal{A}$  is sufficiently non-negligible.

## References

1. J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein. Blake2: simpler, smaller, fast as md5. In *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings 11*, pages 119–135. Springer, 2013.

2. B. Barak and S. Halevi. 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 '05*, page 203–212, New York, NY, USA, 2005. Association for Computing Machinery.
3. D. J. Bernstein et al. Chacha, a variant of salsa20.
4. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge-based pseudo-random number generators. In *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*, pages 33–47. Springer, 2010.
5. M. J. Campagna. Security bounds for the nist codebook-based deterministic random bit generator. Cryptology ePrint Archive, Paper 2006/379, 2006. <https://eprint.iacr.org/2006/379>.
6. S. Coretti, Y. Dodis, H. Karthikeyan, N. Stephens-Davidowitz, and S. Tessaro. On seedless prngs and premature next. *Cryptology ePrint Archive*, 2022.
7. S. Coretti, Y. Dodis, H. Karthikeyan, and S. Tessaro. Seedless fruit is the sweetest: Random number generation, revisited. In *Annual International Cryptology Conference*, pages 205–234. Springer, 2019.
8. J. P. Degabriele, J. Govinden, F. Günther, and K. G. Paterson. The security of chacha20-poly1305 in the multi-user setting. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21*, page 1981–2003, New York, NY, USA, 2021. Association for Computing Machinery.
9. Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs. Security Analysis of Pseudo-Random Number Generators with Input: /Dev/Random is Not Robust. CCS '13, page 647–658, New York, NY, USA, 2013. Association for Computing Machinery.
10. P. Gazi and S. Tessaro. Provably robust sponge-based prngs and kdfs. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*, pages 87–116. Springer, 2016.
11. F. Goichon, C. Lauradoux, G. Salagnac, and T. Vuillemin. *Entropy transfers in the Linux random number generator*. PhD thesis, INRIA, 2012.
12. Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.
13. S. Hirose. Security analysis of drbg using hmac in nist sp 800-90. In K.-I. Chung, K. Sohn, and M. Yung, editors, *Information Security Applications*, pages 278–291, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
14. V. T. Hoang and Y. Shen. Security analysis of nist ctr-drbg. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part I*, pages 218–247. Springer, 2020.
15. A. Luykx, B. Mennink, and S. Neves. Security analysis of blake2’s modes of operation. *IACR Transactions on Symmetric Cryptology*, pages 158–176, 2016.
16. S. Müller. *Documentation and analysis of the linux random number generator*. Federal Office for Information Security, 2020.
17. S. Ruhault. Sok: Security models for pseudo-random number generators. *IACR Transactions on Symmetric Cryptology*, pages 506–544, 2017.
18. T. Shrimpton and R. S. Terashima. A provable-security analysis of intel’s secure key rng. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*,

- Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 77–100. Springer, 2015.
19. T. Shrimpton and R. S. Terashima. Salvaging weak security bounds for blockcipher-based constructions. In *ASIACRYPT (1)*, pages 429–454. Springer, 2016.
20. J. Woodage and D. Shumow. An analysis of nist sp 800-90a. 11477:151–180, 2019.
21. J. Woodage and D. Shumow. An analysis of nist sp 800-90a. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II* 38, pages 151–180. Springer, 2019.
22. K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedtls hmac-drbg. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2020, 2017.