*Journal of The Korea Institute of Information Security & Cryptology* ISSN 1598-3986(Print)
VOL.34, NO.3, Jun. 2024 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2024.34.3.365

# 리눅스 의사난수발생기의 구조 변화 분석*

유 태 일,[1†‡] 노 동 영[2]
[1,2]ETRI 부설연구소 (선임연구원, 책임연구원)

# An Analysis of Structural Changes on the Linux Pseudo Random Number Generator*

Taeill Yoo,[1†‡] Dongyoung Roh[2]
[1,2]The Affiliated Institute of ETRI (Senior Researcher, Principal Researcher)

요 약

모바일 기기나 임베디드 기기 등의 운영체제는 대부분 리눅스 커널을 기반으로 동작한다. 이러한 운영체제는 암호키와 보안 기능 등 시스템 운영에 필요한 난수를 리눅스 커널에 요청한다. 안정적으로 난수를 제공하기 위해, 리눅스 커널은 내부에 전용 의사난수발생기를 탑재하고 있다. 최근에 이 의사난수발생기에 대대적인 구조 변화가 이루어졌다. 그러나 대대적인 변화에도 불구하고 새로운 리눅스 의사난수발생기의 구조에 관한 안전성 분석 결과가 발표되고 있지 않다. 본 논문은 새로운 리눅스 의사난수발생기의 안전성을 분석하기 위한 사전 연구로서 이러한 구조 변경을 살펴본다. 추가로 변경 전과 변경 후의 차이점을 암호학적 관점과 성능 관점으로 구분하여 비교하고 안전성 분석이 필요한 요소를 식별한다. 이 결과는 새로운 리눅스 의사난수발생기 구조에 관한 이해를 제공하여 안전성 분석의 토대로 활용될 수 있다.

ABSTRACT

The operating system (OS) of mobiles or embedded devices is based on the Linux kernel. These OSs request random numbers from the Linux kernel for system operation, such as encryption keys and security features. To provide random numbers reliably, the Linux kernel has a dedicated random number generator (Linux Pseudo Random Number Generator, LPRNG). Recently, LPRNG has undergone a major structural changes. However, despite the major changes, no security analysis has been published on the structure of the new LPRNG. Therefore, we analyze these structural changes as a preliminary study to utilize the security analysis of the new LPRNG. Furthermore, the differences between before and after the changes are divided into cryptographic and performance perspectives to identify elements that require security analysis. This result will help us understand the new LPRNG and serve as a base for security analysis.

**Keywords:** Linux Pseudo Random Number Generator, RNG, Security Analysis

## I. 서 론

리눅스 커널은 현대 컴퓨팅 시스템을 동작시키는 운영체제의 필수 구성요소로 안드로이드, IoT 기기, 임베디드 시스템 등 다양한 운영체제에서 핵심 기능을 담당하고 있다. 이러한 기능 중 하나는 운영체제가 필요로 하는 난수를 제공하는 것으로 이를 위해 리눅스 커널은 전용 난수발생기를 탑재하고 있다. 이

Received(03. 27. 2024), Modified(05. 31. 2024), Accepted(05. 31. 2024)
* 이 논문은 2023년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (2021-0-00046, 국가·공공 정보시스템 안전성 및 활용성 제고를 위한 차세대 암호체계 개발)
† 주저자, taeillyoo@nsr.re.kr
‡ 교신저자, taeillyoo@nsr.re.kr(Corresponding author)

를 리눅스 의사난수발생기(Linux Pseudo Random Number Generator, LPRNG)라 한다. LPRNG는 키 생성과 같은 암호 기능뿐만 아니라 스택 보호 기법 같은 보안 기능과도 밀접한 관련이 있다. 따라서 LPRNG가 안전하게 난수를 생성하지 못하면 암호 키를 예측하거나 보안 기능을 우회하려는 시도 등으로부터 리눅스 기반 시스템의 안전성을 보장할 수 없다[1-10].

리눅스 커널에서 LPRNG의 역할이 중요한 만큼 안전성 분석도 다양한 관점으로 진행되었다. 안전성 분석은 이론 관점과 사용 관점으로 구분할 수 있다.

### 1.1 이론 관점의 안전성 분석 결과

이론적인 분석은 난수발생기의 구조를 모델링하고 이를 증명가능 안전성(provable security) 관점에서 연구한다. B. Barak 등은 난수발생기의 동작을 입력-갱신-출력 세 단계로 나누어 모델링하고 안전성 증명 방법을 제시하였다[1]. 그리고 제시한 증명 방법을 LPRNG에 적용하여 난수 출력 함수 /dev/random이 안전하지 않음을 보였다. Y. Dodis 등은 [1]의 결과를 확장하여 강건성(robustness) 성질을 추가한 분석 방법을 제시하고, 강건성 관점에서 /dev/random을 통한 난수 생성이 안전하지 않음을 증명하였다[2]. 강건성은 난수발생기의 내부상태에 관한 정보를 알고 잡음원을 통제할 수 있는 공격자를 가정할 때도 안전하게 난수를 생성할 수 있는 성질이다.

LPRNG가 가진 구조적인 특징을 분석한 결과들도 다수 발표되었다. Z. Gutterman 등은 최초로 LPRNG의 구조를 체계적으로 분석하였고, 전방향 안전성(forward security) 관점에서 난수 생성 알고리즘의 문제를 발견하였다[3]. 전방향 안전성이란 공격자가 특정 시점에 난수발생기의 내부상태를 획득해도 해당 시점 이전에 생성된 난수를 이상적인 난수와 구별할 수 없다는 성질로, 역추적 저항성(backtracking resistance)이라고도 한다[19]. F. Coichon 등은 LPRNG의 잡음원 수집 단계를 집중적으로 분석하고, /dev/random을 통한 난수 생성이 지연되는 현상의 원인을 규명하였다[4]. P. Lacharme 등은 LPRNG의 각 구성요소를 확률론과 정보이론 관점에서 분석하고, 각 구성요소가 가진 문제점의 개선 방법을 제시하였다[5]. Y. Yeom 등은 LPRNG의 엔트로피 측정기에 관한 이론적 배경이 알려지지 않는 상황에서 LPRNG의 엔트로피 측정 방식이 기반으로 하는 확률분포를 밝히고, 이를 실험을 통해 입증하였다[6].

### 1.2 사용 관점의 안전성 분석 결과

사용 관점의 분석은 LPRNG가 정상 동작하지 않을 때 발생할 수 있는 보안 문제를 연구한다. 그 결과로 안드로이드, IoT 기기 등 리눅스 커널을 기반으로 하는 시스템에서 LPRNG와 관련된 문제가 다수 발견되었다. N. Heininger 등은 여러 대의 서버에서 OpenSSL을 통해 생성한 TLS 인증서와 SSH 공개키를 수집하여 개인키를 찾는 방법을 제시하였다[7]. LPRNG의 엔트로피가 충분하지 않을 때 생성한 난수가 OpenSSL에 내장된 난수발생기의 초깃값으로 사용되는 경우 일부 키 쌍이 노출됨을 보였다. S.H. Kim 등은 안드로이드에서 애플리케이션이 실행될 때 각 애플리케이션에 포함된 OpenSSL의 내부상태가 모두 같은 초깃값을 가지는 현상을 발견하였다[8]. 이 현상은 안드로이드의 자이고트(Zygote)가 애플리케이션을 실행할 때 기존에 생성한 템플릿을 복사하기 때문에 발생한다. 이때 OpenSSL 난수발생기의 내부상태도 템플릿에서 함께 복사된다. 이 상황에서 LPRNG의 취약한 난수가 OpenSSL의 초기화에 사용되면 애플리케이션이 가진 OpenSSL 내부상태를 예측할 수 있음을 보였다. D. Kaplan 등은 부팅 초기의 엔트로피 부족으로 스택 보호 기법의 한 종류인 스택 카너리(stack canary) 값을 예측하여 이를 우회할 수 있음을 보였다[9]. T. Yoo 등은 리눅스 커널 기반 IoT 운영체제 브릴로(brillo)를 대상으로 부팅 초기 엔트로피가 부족한 상황에서 SSL 통신 초기의 사전 공유비밀(pre-master secret)을 높은 확률로 예측할 수 있음을 보였다[10].

LPRNG의 중요성만큼 다양한 관점에서 분석이 진행되었고, 여러 가지 문제점이 밝혀졌다. 이러한 문제점을 해결하기 위해 LPRNG는 최근 대대적으로 구조를 변경하였다. 이 변화는 리눅스 커널 5.17(2022년 3월 20일 배포)부터 확인할 수 있다. 엔트로피 수집 방법과 엔트로피 축적 방법, 엔트로피 관리 방식, 난수 생성 방식 등 대부분의 구성요소를 변경하였다.

이런 변화로 새로운 LPRNG에는 기존에 연구된

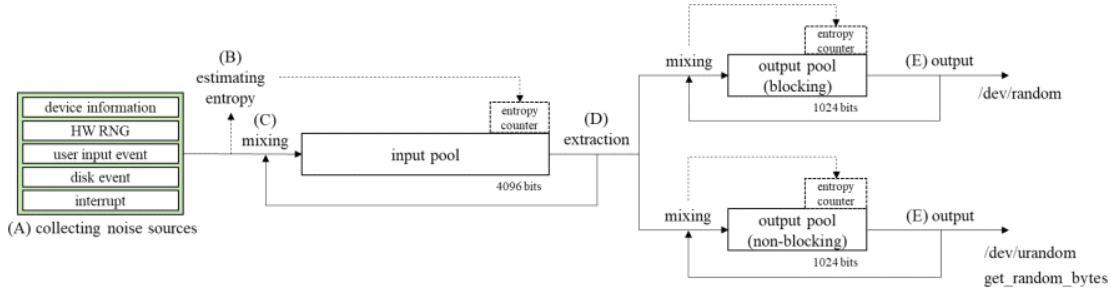


Fig. 1. Overall structure of the previous LPRNG (kernel version 3.10)

안전성 분석 방법을 적용하기 어려워졌다. 그럼에도 새로운 구조에 대한 안전성 연구 결과가 발표되지 않고 있다. 독일 연방정보기술보안청(Bundesamt für Sicherheit in der Informationstechnik, BSI)이 새로운 LPRNG에 대한 보고서를 주기적으로 업데이트하고 있으나 평가 관점에서만 기술하고 있고, 암호학적 배경과 안전성에 대해서는 언급하지 않는다[11]. 이를 제외하면 LPRNG의 구조 변경에 대한 이론적인 배경이나 이를 뒷받침하는 자료를 찾기 어려운 상황이다. 따라서 본 논문에서는 LPRNG의 변경 전후를 비교하고 변경 사항을 분석하여 새로운 LPRNG의 안전성 연구의 토대를 마련하고자 한다.

## II. 리눅스 의사난수발생기(LPRNG)의 구조

ISO/IEC 18031:2011[12]은 국제 표준 난수발생기 명세이다. [12]가 정의한 난수발생기 기능 모델(RBG functional model)에 따르면 난수발생기는 내부상태를 가지며, 잡음원 입력 - 내부상태 갱신 - 출력 생성 단계로 난수를 생성한다. 변경 전과 변경 후 LPRNG의 구조는 모두 국제 표준을 따르고 있어 난수발생기 기능 모델을 기준으로 볼 때 변경 요소가 없어 보인다. 또한 사용자 관점에서 난수를 요청하는 방식(/dev/random, /dev/urandom/, get_random_bytes)도 변경되지 않았다. 그러나 단계별 구성요소를 살펴보면 세부적인 구조가 모두 변경된 것을 확인할 수 있다. 이번 장은 변경 전과 후의 LPRNG의 구조를 설명한다.

### 2.1 변경 전 LPRNG의 구조

변경 전 LPRNG[1]는 세 개의 엔트로피 풀을 가진다. 한 개는 잡음원을 수집하고 축적하는 4096비트 크기의 풀로 입력 풀(input pool)이라 한다. 나머지 두 개는 축적된 잡음원을 바탕으로 난수를 출력하는 1024비트 크기의 풀로 출력 풀(output pool)이라 한다.

LPRNG는 리눅스 커널에서 발생하는 키보드 입력이나 마우스 움직임, 디스크 입출력(I/O) 시간, 인터럽트 정보 등 예측 불가능한 현상을 잡음원으로 활용한다. 예를 들어 키보드에 입력이 발생하면 입력된 키의 유형, 입력 시간 같은 값이 잡음원이 되고, 디스크에 이벤트가 발생하면 발생 시간과 디스크 번호가 잡음원이 된다. 잡음원 수집이란 이러한 값을 획득하는 동작을 의미한다.

수집한 잡음원은 입력 풀에 혼합되고, 단계별 절차에 따라 난수를 생성한다. Fig. 1은 변경 전 LPRNG의 구조를 나타내며, Fig. 1의 (A)부터 (E)까지 동작 과정은 다음과 같다.

(A)는 운영체제에서 발생하는 예측 불가능한 동작을 감지하여 잡음원을 수집하는 단계이다. 디바이스 정보, CPU 탑재 난수발생기, 사용자 입력 정보, 디스크 입출력(I/O), 인터럽트의 다섯 가지 동작을 기반으로 잡음원을 수집한다(새로운 LPRNG는 10가지로 확장하였음).

(B)는 수집한 잡음원의 엔트로피를 측정하는 단계이다. 측정한 엔트로피 값은 입력 풀에 연결된 엔트로피 카운터에 더한다. 엔트로피 측정 방법은 잡음원 종류에 따라 달라지며, 어떤 잡음원은 사전에 정의된 값을 적용하고, 어떤 잡음원은 엔트로피 측정 방식을 적용한다. 구체적인 내용은 2.3절에서 설명한다.

1) 변경 전 LPRNG는 리눅스 커널 3.10(2013년 6월 30일 배포)을 기준으로 분석하였다. 이는 기존에 발표된 LPRNG 분석 결과가 참고한 가장 최신 버전이며, 4.7까지 구조적인 차이는 없다. 4.8에서 5.16까지 출력 함수에 ChaCha20을 적용한 변경 사항이 있었으나 그 외의 다른 변경은 없다.

(C)는 수집한 잡음원을 입력 풀에 혼합하는 단계이다. 잡음원을 32비트 단위로 나누고 혼합 함수(mix_pool_bytes)를 통해 32비트 워드 하나씩 혼합한다(2.1.1 참고). 잡음원이 입력될 때마다 이 단계를 통해 입력 풀을 갱신한다.

(D)는 시딩(seeding) 단계이다. 시딩은 입력 풀에 축적된 잡음원을 이용하여 시드를 추출하고 이를 출력 풀에 전달하는 동작이다.[2] 시딩 이후 입력 풀의 엔트로피 카운터는 전달한 시드의 크기만큼 감소하고, 출력 풀의 엔트로피 카운터는 시드의 크기만큼 증가한다. 시딩 알고리즘은 2.1.2에서 자세하게 다룬다.

(E)는 난수 생성 단계이다. 사용되는 출력 풀은 난수 요청 방식에 따라 달라진다. /dev/random으로 난수를 요청하는 경우 차단형(blocking) 출력 풀에서 난수를 생성하고, /dev/urandom이나 get_random_byte로 난수를 요청하는 경우 비차단형(non-blocking) 출력 풀에서 난수를 생성한다.

차단형 출력 풀은 엔트로피 카운터가 기준값인 192비트 이상일 때 난수를 출력하고, 기준값 이하라면 입력 풀에 시드를 요청하여 엔트로피 카운터를 기준값 이상으로 높인 이후 난수를 출력한다. 이러한 특징으로 엔트로피 카운터 값에 따라 난수 생성이 지연될 수 있다. 반대로 비차단형 출력 풀은 엔트로피 카운터와 관계없이 즉시 난수를 출력한다. 차단형과 비차단형 출력 풀은 모두 출력한 난수의 크기만큼 엔트로피 카운터를 감소시킨다. 난수 출력 알고리즘은 2.1.2에서 자세하게 다룬다.

### 2.1.1 혼합 함수

혼합 함수(mixing function)는 잡음원을 입력 풀에 추가하거나 시드를 출력 풀에 전달할 때 사용한다. Table 1은 시드를 출력 풀로 전달할 때 적용하는 혼합 함수 mix_pool_bytes의 동작 절차이다. 입력 풀에서 출력 풀로 전송한 시드는 32비트 워드 단위로 나뉘어서 입력된다. d는 32비트 크기씩 입력된 시드를 의미하고, pool은 혼합 함수를 적용하는 엔트로피 풀을 의미한다. 여기서는 출력 풀을 의미한다.

잡음원 d가 입력되면 데이터가 혼합될 위치 i를 계산한다. i는 출력 풀 구조체에 포함된 변수로 잡음원이 추가될 때마다 1씩 감소한다. 다음으로 출력 풀에서 인덱스가 i, i+1, i+7, i+14, i+20, i+26인 원소를 선택하여 XOR 한다. 마지막으로 현재까지 계산된 데이터 tmp의 하위 3비트를 이용하여 table의 원소를 선택하고, (tmp 〉〉 3)과 XOR 한다. 이 값을 pool의 i번째 원소에 저장한다.

Table 1. Pseudocode of mix_pool_bytes

```
mix_pool_bytes(pool, d)
    i = pool.i = (pool.i - 1) mod 32
    tmp = d
    tmp = tmp ⊕ pool[i]
    tmp = tmp ⊕ pool[(i+1) mod 32]
    tmp = tmp ⊕ pool[(i+7) mod 32]
    tmp = tmp ⊕ pool[(i+14) mod 32]
    tmp = tmp ⊕ pool[(i+20) mod 32]
    tmp = tmp ⊕ pool[(i+26) mod 32]
    tmp = (tmp 〉〉 3) ⊕ table[tmp & 7]
    pool[i] = tmp

table[8] = {0x00000000, 0x3b6e20c8,
            0x76dc4190, 0x4db26158,
            0xedb88320, 0xd6d6a3e8,
            0x9b64c2b0, 0xa00ae278}
```

출력 풀에서 선택하는 원소의 인덱스는 TGFSR(Twisted Generalized Feedback Shift Register)[13]에서 사용하는 다항식에 기반한다. TGFSR이 생성하는 출력값의 주기는 다항식에 의해 결정되며, 출력값의 주기를 최대로 만드는 성질을 가진 다항식을 선택한다. 이러한 다항식은 여러 개 존재하는데 출력 풀 갱신에는 다항식 $x^{32}+x^{26}+x^{20}+x^{14}+x^{7}+x+1$을 적용한다.

입력 풀의 갱신에 적용한 다항식은 $x^{128}+x^{103}+x^{76}+x^{51}+x^{25}+x+1$이다. 입력 풀에 적용하는 혼합 함수는 TGFSR 상에서 선택하는 원소의 인덱스에만 차이가 있고, 동작 과정은 출력 풀에 적용하는 혼합 함수와 같다.

### 2.1.2 추출 함수

추출 함수(extract function)는 출력 풀에서 난수를 생성하거나 입력 풀에서 출력 풀로 전달하는 시드를 생성할 때 사용한다. 두 경우 모두 같은 알고리즘을 사용하지만, 각 풀의 크기에 따라 해싱 횟수에 차이가 있다. 해싱에는 해시함수 SHA-1을 적용한

2) 변경 전 LPRNG에서는 엔트로피 전송(entropy transfer)이라고 하며, 새로운 LPRNG에서는 시딩으로 표현한다. 본 논문은 모두 시딩으로 부른다.

다. 출력 풀의 크기는 1024비트로 512비트씩 2회의 해싱을 하며, 입력 풀의 크기는 4096비트로 512비트씩 8회의 해싱을 한다. Fig. 2는 출력 풀에 추출 함수를 적용하는 과정이다. Fig. 2의 (A) 단계부터 (C) 단계까지 동작 과정은 아래와 같다.

(A) 단계: 출력 풀의 상위 512비트를 SHA-1으로 해싱하여 160비트 출력을 얻는다. 160비트를 32비트씩 다섯 워드로 나누고 이 중 하나를 j mod 5를 계산하여 선택한다. j는 추출 함수가 실행될 때 0으로 초기화되고 해싱마다 2만큼 증가한다. 선택한 워드는 출력 풀의 위치 i에 입력한다. 여기서 i는 출력 풀 구조체의 변수로 워드 삽입 위치를 가리키고 해싱 연산을 할 때마다 1만큼 감소한다.

(B) 단계: 하위 512비트를 SHA-1으로 해싱하여 160비트를 얻고, 워드 하나를 선택하여 출력 풀의 위치 i-1에 입력한다(입력 위치 1 감소). 출력 풀을 모두 해싱하였으면 i-1에 입력한 값을 i-2에도 입력한다.

(C) 단계: i-2부터 역순으로 512비트를 선택하여 해싱한다. 마지막으로 해싱 결과 160비트를 상위 80비트와 하위 80비트로 나누고 서로 XOR 하여 80비트를 출력한다. 이를 폴딩 연산이라 한다. 이 과정을

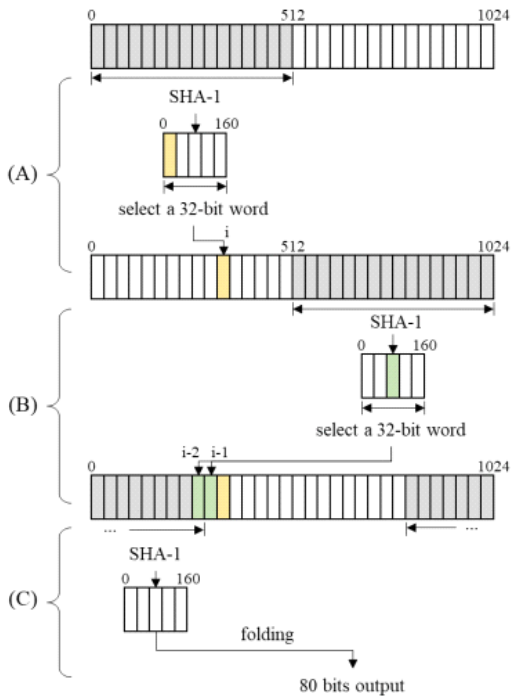


Fig. 2. Working process of the extract function

요청받는 난수의 크기가 될 때까지 반복한다.

## 2.2 새로운 LPRNG의 구조

새로운 LPRNG[3]의 구조는 잡음원 입력 - 내부 상태 갱신 - 출력 생성 단계로 변경 전 LPRNG와 같다. 그러나 단계별 구성요소를 대대적으로 변경하였다. 이 절에서는 새로운 LPRNG의 동작 과정과 변경된 구성요소인 엔트로피 풀의 개수와 크기, 잡음원 혼합 알고리즘, 시드 추출 알고리즘, 난수 생성 알고리즘을 설명한다.

엔트로피 풀은 입력 풀, 부모 CRNG(base Cryptographic Random Number Generator), 자녀 CRNG(secondary CRNG), 인터럽트 잡음원 전용 풀(fast pool)로 구성된다. 입력 풀은 256비트 크기로 하나이며, 부모 CRNG도 256비트 크기로 하나이다. 자녀 CRNG는 256비트 크기이며, CPU가 가진 코어의 개수만큼 가지고 있다. 또한 인터럽트 잡음원 전용 풀도 CPU의 코어 개수만큼 가지며, 풀의 크기는 CPU의 아키텍처에 따라 128비트(32비트 프로세서)이거나 256비트(64비트 프로세서)이다.[4] Fig. 3은 새로운 LPRNG의 구조를 나타낸다. Fig. 3의 (A)부터 (G)까지 동작 과정은 다음과 같다.

(A)는 잡음원 수집 단계이다. 변경 전 LPRNG가 사용한 다섯 가지 잡음원에 새로운 유형의 잡음원 다섯 가지를 추가하였다.

(B)는 엔트로피 측정 단계이다. 잡음원의 엔트로피는 2.3의 방법으로 측정하고 측정 결과를 엔트로피 추정기(entropy esimator)[5]에 더한다. 변경 전 LPRNG와 같이 어떤 잡음원은 사전에 정의된 값을 적용하며, 어떤 잡음원은 엔트로피 측정 방식을 적용한다. 엔트로피 측정 방법은 2.3절에서 자세하게 다룬다. 새로운 LPRNG의 경우 엔트로피 추정기의 값은 증가만 하고, 최댓값인 256에 도달하면 더 이상 계산하지 않는다.

(C)는 잡음원을 입력 풀에 혼합하는 단계이다. 새로운 LPRNG는 입력 풀의 크기를 256비트로 변경

3) 새로운 LPRNG는 분석 당시 최신 버전인 리눅스 커널 6.5(2023년 8월 27일 배포)을 기준으로 분석하였다.
4) 자녀 CRNG와 전용 풀은 DEFINE_PER_CPU 매크로를 통해 CPU의 각 코어에 할당된다.
5) 엔트로피 추정기와 엔트로피 카운터는 이름이 다를 뿐 동작 방식은 변경 없다.

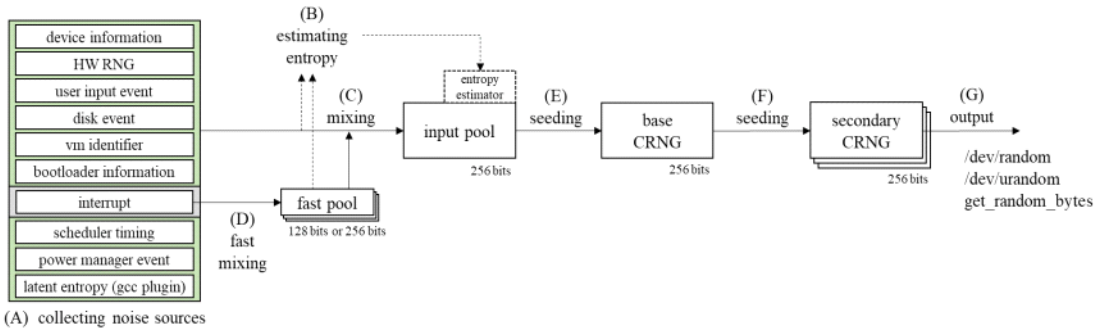


Fig. 3. Overall structure of the new LPRNG (kernel version 6.5)

하였다. 변경 전 LPRNG의 입력 풀 크기인 4096 비트 대비 1/16로 줄었다. 혼합 함수는 TGFSR 기반에서 해시함수 blake2s[14] 기반으로 변경하였다. 이에 따라 잡음원을 32비트 워드로 나누어 처리하지 않고 한 번에 처리한다. 혼합 함수는 2.2.1에서 자세하게 다룬다.

(D)는 인터럽트 잡음원을 처리하는 단계이다. 인터럽트는 CPU의 코어마다 발생하는 잡음원으로 CPU의 코어마다 자신의 인터럽트 잡음원 전용 풀(fast pool)을 가지고 있다. 전용 풀의 크기는 CPU 아키텍처에 따라 128비트(32비트 프로세서) 또는 256비트(64비트 프로세서)로 결정된다. 인터럽트 잡음원이 입력되면 해시함수 SipHash[15]를 적용하여 전용 풀에 혼합한다. SipHash의 출력 크기는 32비트 CPU의 경우 64비트, 64비트 CPU의 경우 128비트로 동작한다. 구체적인 과정은 2.2.2에서 다룬다.

(E)는 부모 CRNG에 시드를 전달하는 단계로 시딩이라 부른다. 부모 CRNG에 대한 시딩이란 입력 풀에서 시드를 추출하여 부모 CRNG에 전달하는 절차를 의미한다. 부모 CRNG의 내부상태는 ChaCha20[16]의 512비트 내부상태 중 256비트 키에 해당한다. 전달받은 시드는 ChaCha20의 키(부모 CRNG의 내부상태)로 사용한다. 부모 CRNG는 엔트로피 추정기를 가지고 있지 않아 시딩은 엔트로피 측정 없이 이루어진다. 자세한 과정은 2.2.3에서 다룬다.

(F)는 부모 CRNG에서 자녀 CRNG로 시딩하는 단계이다. 자녀 CRNG는 CPU의 코어 개수만큼 존재하기 때문에 부모 CRNG는 자녀 CRNG의 개수만큼 시드를 생성한다. 생성한 시드는 자녀 CRNG의 내부상태(ChaCha20의 256비트 키)로 입력한다. 자녀 CRNG도 엔트로피 추정기가 없어 엔트로피 측정 없이 시딩이 이루어진다. 자세한 과정은 2.2.4에서 다룬다.

(G)는 난수 생성 단계이다. CPU의 코어는 자신에게 대응된 자녀 CRNG에 난수를 요청한다. 자녀 CRNG는 요청받은 난수의 크기만큼 ChaCha20을 반복 사용하여 난수를 생성한다. 변경 전 LPRNG의 경우 차단형 출력 풀과 비차단형 출력 풀을 구분하여 난수를 생성하였으나 새로운 LPRNG는 차단 기능을 비활성화하여 차단/비차단 구분 없이 난수를 생성한다. 즉, 사용자의 난수 요청 방식 세 가지(/dev/random, /dev/urandom, get_random_bytes)는 동작 과정이 같아졌다. 난수 요청 및 출력 절차는 소절 2.2.5에서 다룬다.

### 2.2.1 혼합 함수

혼합 함수는 잡음원을 입력 풀에 추가하여 입력 풀의 내부상태를 갱신한다. 입력 풀의 크기를 256비트로 변경하였는데 입력 풀의 내부상태를 해시함수 blake2s의 내부상태로 대체하였기 때문이다. 그 결과 혼합 함수의 동작은 blake2s의 갱신 함수(blake2s update)를 적용하는 방식으로 변경되었다.

blake2s의 갱신 함수는 잡음원이 입력될 때마다 동작하여 입력 풀의 상태를 변경시키며 잡음원을 축적한다. Fig. 4는 혼합 함수가 동작하는 과정의 예이다. 변경 전 LPRNG가 잡음원을 32비트 워드 단위로 나누어 혼합하는 것에 비하여 새로운 LPRNG는 잡음원을 나누지 않고 혼합한다.

혼합 함수는 blake2s의 출력 함수(blake2s final)을 통해 256비트 해시값을 생성한다. 이 해시

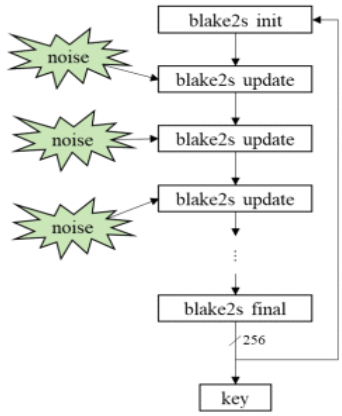


Fig. 4. An example of the mixing function

값은 부모 CRNG에 전송할 시드를 생성하기 위한 키로 사용되고, 입력 풀(blake2s의 내부상태)을 초기화(blake2s init)하는 데 사용된다. 출력된 해시 값의 크기는 256비트이다.

### 2.2.2 인터럽트 전용 풀에 대한 혼합 함수

컴퓨터 구조가 멀티 코어 환경으로 변하면서 코어마다 발생하는 인터럽트 잡음원 처리에 과부하가 발생할 수 있다. 이를 개선하고자 코어별로 인터럽트 잡음원을 처리하기 위한 작은 크기의 입력 풀을 할당한다. 이를 인터럽트 잡음원 전용 풀(fast pool)이라 하며, 크기는 CPU 아키텍처에 따라 128비트(32비트 프로세서) 또는 256비트(64비트 프로세서)로 결정된다.

인터럽트 잡음원은 해시함수 SipHash를 적용하여 전용 풀에 혼합한다. SipHash의 출력은 32비트 CPU의 경우 64비트, 64비트 CPU의 경우 128비트이다. 전용 풀에 누적된 잡음원은 1) 인터럽트 잡음원 혼합 횟수가 1024회에 도달하거나 2) 입력 풀에 혼합 이후 1초가 경과하는 경우 입력 풀에 혼합한다.

### 2.2.3 부모 CRNG에 대한 시딩

부모 CRNG에 대한 시딩은 입력 풀에서 생성한 256비트 시드를 부모 CRNG로 전달하는 동작을 의미한다. 시드 생성 과정을 추출(extraction)이라 한다. Fig. 5는 시드 추출 과정을 나타내며, Fig. 5의 (A)부터 (D)까지의 추출 과정은 다음과 같다.

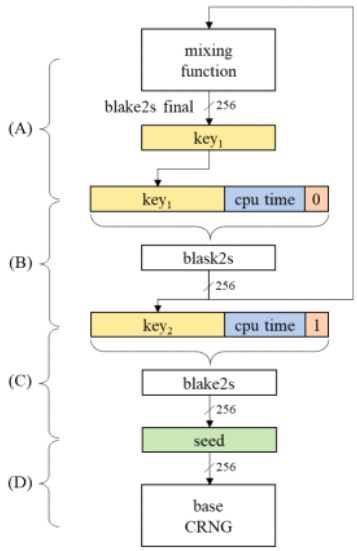


Fig. 5. Seeding process for the base CRNG

(A): 잡음원이 축적된 입력 풀에 blake2s의 출력 함수(blake2s final)를 적용하여 256비트 키($key_1$)를 생성한다.

(B): (A)에서 생성한 256비트 키에 CPU 타이머에서 획득한 값 64비트와 0으로 초기화된 카운터 64비트를 연접한다. 이 값을 blake2s에 입력하여 256비트 키($key_2$)를 생성한다. 이 키는 혼합 함수 내의 blake2s 초기화 함수(blake2s init)의 입력과 키 추출 과정 (C)의 입력으로 사용된다.

(C): (B)에서 생성한 256비트 키에 64비트 CPU 시간, 64비트 카운터를 연접한다. 64비트 CPU 시간은 (B)에서 사용한 값을 다시 사용하고, 카운터는 1만큼 증가한 값이다. 이 값을 blake2s에 입력하여 256비트 시드를 생성한다.

(D): 시드를 부모 CRNG에 입력한다.

변경 전 LPRNG는 난수 생성을 요청받을 때 입력 풀과 출력 풀의 엔트로피 카운터를 비교한 후 시딩하였다. 새로운 LPRNG에서 부모 CRNG에 대한 시딩은 부팅 완료 이후에 60초 주기로 이루어진다.

부팅 완료 전에는 수집한 잡음원이 부족하여 LPRNG를 세 가지 상태로 구분하여 시딩한다. 엔트로피 추정기의 값이 0~127비트인 경우, “EMPTY” 상태, 128~255비트인 경우 “EARLY” 상태, 256비트 이상인 경우 “READY”로 관리한다. 부팅 초기에는 엔트로피 추정기 값이 매우 낮아 EMPTY 상태이다. 이 상태에서는 잡음원이 입력될

때마다 시딩한다. EARLY 상태의 경우 잡음원이 입력되면 입력 풀의 상태만 갱신하며 시딩을 하지 않는다. READY 상태의 경우 60초 주기로 시딩한다. 엔트로피 카운터의 값이 256에 도달한 이후에는 READY 상태가 유지되며, 엔트로피 추정기를 비활성화한다.[6)]

#### 2.2.4 자녀 CRNG에 대한 시딩

부모 CRNG의 내부상태는 ChaCha20의 512비트 내부상태 중 256비트 키에 해당한다. 2.2.3의 시딩 과정을 통해 입력한다. ChaCha20의 나머지 256비트는 ChaCha20의 초기화 절차에 따라 설정한다. 즉, 128비트는 초기화 상수, 96비트는 논스(nonce), 32비트는 카운터로 설정한다. LPRNG에서는 논스와 카운터를 0으로 초기화한다.

Fig. 6은 자녀 CRNG에 대한 시딩 과정이다. 부모 CRNG는 n번째 내부상태에서 시작한다고 가정하였다. 먼저 부모 CRNG의 n번째 내부상태에 ChaCha20을 적용하여 512비트 출력을 생성한다. 최상위 256비트는 부모 CRNG의 n+1번째의 키로 사용하고, 나머지 256비트는 자녀 CRNG의 시드로 입력한다. 자녀 CRNG는 CPU 개수만큼 존재하기 때문에 부모 CRNG의 내부상태를 갱신하며 256비트씩 시딩한다.

자녀 CRNG에 대한 시딩은 부모 CRNG의 시딩 횟수와 자녀 CRNG의 시딩 횟수를 비교하여 이루어진다. 자녀 CRNG의 시딩 횟수가 부모 CRNG의 시딩 횟수가 보다 적은 경우 시딩된다. 그 외에는 시딩이 발생하지 않는다.

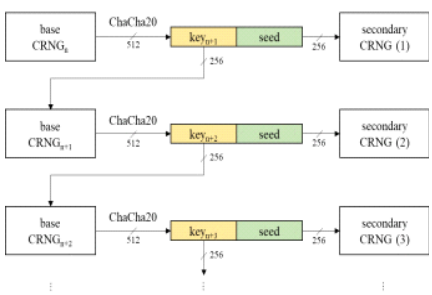


Fig. 6. Seeding process for secondary CRNGs

6) 소스 코드에서는 EMPTY, EARLY, READY 플래그로 LPRNG의 상태를 확인하며, 이는 crng_ready() 매크로가 반환하는 값으로 점검할 수 있다.

#### 2.2.5 난수 출력

난수 출력은 부모 CRNG가 자녀 CRNG에 대한 시드를 생성하는 과정과 같은 방식으로 동작한다. 자녀 CRNG의 내부상태에 ChaCha20을 반복적으로 적용하여 512비트 단위로 난수를 생성한다. Fig. 7은 난수를 생성하는 과정이다.

자녀 CRNG가 난수 생성 요청(request n)을 받으면 자신의 내부상태를 이용하여 512비트 출력을 생성한다. 최상위 256비트($key_{n+1}$)는 다음 난수 요청을 위해 저장하고, 나머지 256비트를 난수로 출력한다. 이후부터는 요청받은 크기의 난수가 될 때까지 ChaCha20을 반복하여 512비트 단위로 난수를 생성한다. 다음 난수 요청(request n+1)을 받으면 이 과정을 반복한다.

자녀 CRNG에 대한 시딩은 부모 CRNG와 자녀 CRNG에 대한 시딩 횟수를 비교하여 이루어진다. 자녀 CRNG는 난수 생성 요청을 받은 후 자신의 시딩 횟수와 부모 CRNG의 시딩 횟수를 비교한다. 이때 자신의 시딩 횟수가 부모 CRNG의 시딩 횟수보다 적으면 시딩이 발생한다. 시딩이 발생하지 않았다면 다음 난수 요청을 위해 저장된 키($key_{n+1}$, $key_{n+2}$, ...)를 이용하여 요청받는 크기의 난수를 생성한다.

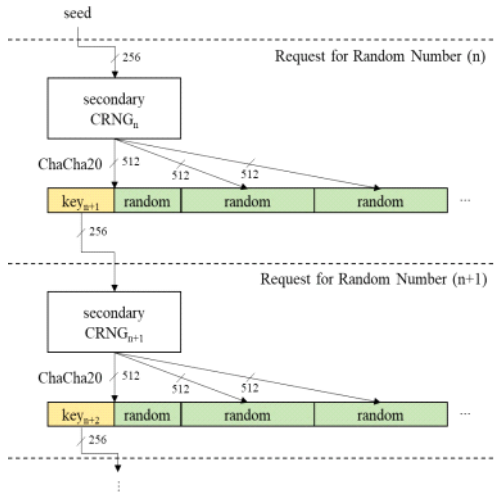


Fig. 7. Process of outputting random numbers

### 2.3 잡음원

잡음원의 엔트로피는 두 가지 방식으로 측정한다.

첫 번째는 사전에 정의된 고정된 값을 적용하여 엔트로피를 계산하는 방식이며, 두 번째는 잡음원 발생 시간을 이용하여 엔트로피를 측정하는 방식이다. 마지막 잡음원 발생 시간과 직전의 잡음원 발생 시간의 차이를 이용하여 엔트로피를 측정한다. 자세한 과정은 2.3.1에서 다룬다.

대부분의 잡음원은 첫 번째 방식으로 엔트로피를 측정하고, 사용자 입력과 디스크 이벤트 잡음원의 엔트로피는 두 번째 방식으로 엔트로피를 측정한다.

### 2.3.1 잡음원의 엔트로피 측정

LPRNG는 엔트로피를 측정하는 고유한 방식을 가지고 있다. 잡음원이 발생한 시간 간격을 기반으로 엔트로피를 측정한다. $t_0, t_1, t_2, \cdots$를 잡음원이 발생한 시간이라 하자. 그러면 $i$번째 발생한 잡음원의 엔트로피 $H$는 다음과 같이 측정한다.

i) $\Delta_i^1 = t_i - t_{i-1}, i = 1, 2, 3, \cdots,$

ii) $\Delta_i^2 = \Delta_i^1 - \Delta_{i-1}^1, i = 2, 3, 4, \cdots,$

iii) $\Delta_i^3 = \Delta_i^2 - \Delta_{i-1}^2, i = 3, 4, 5, \cdots,$

iv) $d_i = \lfloor log_2(\min\{|\Delta_i^1|, |\Delta_i^2|, |\Delta_i^3|\}) \rfloor,$

v) $H = \min\{d_i, 11\}.$

$H = \min\{d_i, 11\}$는 $d_i$가 11을 초과하는 경우 11로 제한함을 의미한다. $\lfloor x \rfloor$의 값은 $x$보다 작거나 같은 가장 큰 정수이다. 측정된 $H$는 입력 풀의 구조체에 포함된 엔트로피 추정기에 더해진다(Fig. 3의 (B) 단계).

### 2.3.2 잡음원 종류

최신 LPRNG는 10가지 잡음원을 수집한다(Table 2). 다섯 가지는 변경 전 LPRNG부터 활용하던 잡음원이고, 나머지는 새로운 LPRNG에 추가된 잡음원이다(Table 2의 파란 상자). 여기서는 어떤 특징을 잡음원으로 활용하는지와 잡음원별 엔트로피 측정 방법을 중심으로 정리한다.

1) 기기 정보(device information)

기기 정보는 디바이스 드라이버의 고유 정보를 기반으로 하는 잡음원이다. 일련번호 같이 기기마다 다른 값이다. 부팅이 시작될 때 0으로 초기화된 입력 풀의 상태를 변경하는 역할을 한다. 그러나 고정된 값이기 때문에 디바이스 드라이버가 초기화될 때 한 번만 사용한다. add_device_randomness 함수를 통해 입력 풀에 혼합되고, 엔트로피는 0으로 간주한다.

2) CPU 탑재 하드웨어 난수발생기(HW RNG)

CPU 제조사는 자신의 칩에 하드웨어 기반 난수발생기를 탑재한다. 대표적인 예는 인텔 x86의 RDRAND 명령어를 통한 난수 생성 방식이다. CPU 탑재 하드웨어 난수발생기에서 생성한 난수는 add_hwgenerator_randomness 함수를 통해 입력 풀에 혼합된다.

엔트로피는 CPU 제조사가 제공하는 고정된 값을 사용한다. 이 값은 난수 1024비트당 엔트로피를 의미한다. 예를 들어, Freescale i.MX CPU는 고정값 19를 사용하고 비트당 엔트로피는 19/1024 = 0.019로 계산한다(참고: CPU 제조사별 엔트로피 고정값은 리눅스 커널 소스 코드의 /drivers/char/hw_random 위치에서 확인 가능).

3) 사용자 입력 이벤트(user input event)

사용자 입력은 키보드나 마우스, 터치스크린, 게임용 컨트롤러 등 다양한 장치의 입력을 기반으로 하는 잡음원이다. add_input_randomness 함수를 통해 입력 풀에 혼합한다. 엔트로피는 2.3.1에서 기술한 LPRNG의 엔트로피 측정 방법으로 계산하며, 최소 0부터 최대 11까지 가능하다.

4) 디스크 장치 이벤트(disk event)

디스크 장치 이벤트는 디스크 입출력 시간을 기반으로 하는 잡음원이다. 디스크가 여러 개인 경우는 디스크마다 이벤트 발생 시간을 별도로 관리한다. add_disk_randomness 함수를 통해 입력 풀에 혼합한다. 엔트로피는 2.3.1에서 기술한 방법으로 계산하며, 최소 0부터 최대 11까지 가능하다.

5) 가상머신 식별자(vm identifier)

가상머신 스냅숏(snapshot)을 복사해서 사용하는 경우 여러 개의 스냅숏이 같은 LPRNG의 내부 상태를 가지는 문제가 발생한다. 이런 이유로 같은 스냅숏이더라도 다른 난수를 생성할 수 있도록 가상머신 식별자를 잡음원으로 입력한다. add_vmfork_randomness 함수를 통해 입력 풀에 혼합하고, 엔

트로피는 0으로 간주한다.

6) 부트로더 정보(bootloader information)

부트로더 정보는 부팅 설정 정보에 대한 문자열을 기반으로 하는 잡음원이다. CPU 유형, 파일시스템 설정 등의 부팅 관련 정보를 문자열 형식으로 add_bootloader_randomness 함수를 통해 입력 풀에 혼합한다. 엔트로피는 신뢰 부팅(trust boot)이 설정되면 문자열 길이(바이트 단위) × 8로 계산하고, 그 외에는 0으로 간주한다.

7) 인터럽트(interrupt)

CPU의 각 코어는 동작 중에 여러 가지 인터럽트를 처리한다. 인터럽트 잡음원은 인터럽트가 발생한 시간을 기반으로 한다. 멀티 코어 환경에서 인터럽트는 여러 개의 코어에서 발생한다. 인터럽트 잡음원은 코어마다 별도로 할당된 인터럽트 잡음원 전용 풀(fast pool)에 수집된다. 전용 풀에 축적된 인터럽트 잡음원은 1초 주기 또는 인터럽트 횟수가 1024회를 초과하는 경우 add_interrupt_randomness를 통해 입력 풀에 혼합된다. 엔트로피는 인터럽트 발생 횟수를 64로 나눠 정수 부분만 취하며(소수점 버림), 최솟값은 1로, 최댓값은 64 또는 128로 제한한다(CPU 아키텍처별 SipHash 출력 크기).[7]

8) 스케줄러 동작 특성(scheduler timing)

프로세스는 커널의 스케줄러에 의해 문맥 교환을 하며 동작한다. 스케줄러가 문맥 교환을 시작하고 종료하기까지 걸리는 시간은 상황에 따라 차이를 보인다. 스케줄러 동작 특성 잡음원은 이러한 현상에서 발생하는 시간을 기반으로 한다. 프로세스가 종료되고 다른 프로세스로 문맥 교환되기 직전의 시간을 wait_for_random_bytes 함수를 통해 입력 풀에 혼합하며, 엔트로피는 1로 간주한다.

9) 전원 관리 이벤트(power manager event)

전원 관리 장치에 발생한 이벤트 유형을 기반으로 하는 잡음원이다. random_pm_notification 함수를 통해 입력 풀에 혼합하고, 엔트로피는 0으로 간주한다.

10) 지연 엔트로피(latent entropy)

리눅스 커널을 컴파일할 때 여러 가지 플러그인을 추가할 수 있다. 지연 엔트로피는 컴파일 옵션에 "LATENT_ENTROPY_PLUGIN"을 설정하면 사용할 수 있는 잡음원이다. 멀티 코어 환경에서 코어마다 함수 실행 속도 차이가 발생하는데 LPRNG는 이를 지연(latent)이라 부른다. CPU의 코어마다 지연 엔트로피를 확보하기 위해 각 코어에 할당된 지역변수 __latent_entropy에 값을 저장하고, 여러 개의 코어가 전역변수 latent_entropy를 갱신한다. latent_entropy에 저장된 값은 add_latent_entropy 함수를 통해 입력 풀에 혼합하고, 엔트로피는 0으로 간주한다.

Table 2. Noise sources of the new LPRNG

| type (function name) | entropy(e) |
|---|---|
| device information (add_device_randomness) | 0 |
| HW RNG (add_hwgenerator_randomness) | fixed value for each CPU |
| user input event (add_input_randomness) | $0 \le e \le 11$ |
| disk event (add_disk_randomness) | $0 \le e \le 11$ |
| vm identifier (add_vmfork_randomness) | 0 |
| bootloader information (add_bootloader_randomness) | if trust boot, data size × 8 else 0 |
| interrupt (add_interrupt_randomness) | $1 \le e \le 64$ (32-bit CPU) $1 \le e \le 128$ (64-bit CPU) |
| scheduler timing (wait_for_random_bytes) | 1 |
| power manager event (random_pm_notification) | 0 |
| latent entropy (add_latent_entropy) | 0 |

* blue box: new noise sources

7) 소스 코드에서 clamp_t 함수를 이용하여 계산하는데 clamp_t는 입력값의 범위를 제한하는 함수이다. 입력값이 최솟값을 벗어나는 경우 최솟값을, 최댓값을 벗어나는 경우 최댓값을 반환한다. LPRNG는 최솟값을 1, 최댓값을 전용 풀의 크기(64 또는 128)로 설정하여 엔트로피가 0으로 측정되면 1을 반환하고, 전용 풀의 크기를 벗어나면 전용 풀의 크기를 반환한다.

위에서 설명한 10가지 잡음원의 종류와 혼합에 사용된 함수 이름, 해당 잡음원의 엔트로피 측정값을 요약하면 Table 2와 같다. 측정값 e는 정수이다.

## III. 주요 구조 변화 비교

이번 장은 2장의 분석 결과를 바탕으로 LPRNG의 구조 변화를 비교하고, 이를 통해 확인한 일곱 가지 변경 사항을 암호학적 관점과 성능 관점으로 구분하여 설명한다. Table 3은 변경 전후의 차이점을 요약한 것이며 이번 장에서 각 항목을 구체적으로 기술한다.

Table 3. Structural changes: old vs. new LPRNG

| cryptographic perspective | | |
|---|---|---|
| **features** | **old LPRNG** | **new LPRNG** |
| mixing function | TGFSR | hash function (blake2s) |
| seeding algorithm | dedicated method (SHA-1) | HKDF-like method (blake2s) |
| key change cycle | if entropy counter is less than 192 bits | every 60 seconds |
| noise source types | 5 types | 10 types (5 old+5 new) |
| **performance perspective** | | |
| **features** | **old LPRNG** | **new LPRNG** |
| blocking mode | whenever entropy counter is less than 192 bits | no blocking since READY state |
| the number of CRNGs | none (instead, there are two output pools) | one for each CPU core |
| the number of fast pools | none | one for each CPU core |

### 3.1 암호학적 관점에서 구조 변화

1) 잡음원 혼합 방식 변경

변경 전 LPRNG는 잡음원 혼합 함수에 TGFSR을 사용하였으나 새로운 LPRNG는 해시함수 blake2s의 갱신 함수를 사용한다. TGFSR의 경우 공격자가 LPRNG의 내부상태를 알게 되면 이전의 상태를 복원할 수 있어 전방향 안전성을 만족하지 못하는 것으로 알려졌다[3].

새로운 LPRNG는 이러한 문제점을 해결하기 위해 해시함수를 혼합 함수로 사용한다. 즉, 혼합 함수는 해시함수의 역상 저항성(pre-image resistance)에 기반하여 전방향 안전성을 제공한다. 그러나 LPRNG의 전체가 전방향 안전성을 제공하는지는 알려지지 않아 추가적인 연구가 필요해 보인다.

2) 부모 CRNG에 대한 시딩 방식 변경

변경 전 LPRNG는 2.1.2에서 언급한 것과 같이 SHA-1 기반의 추출 함수를 사용하여 생성한 시드를 출력 풀로 전송하였다. 현재 SHA-1은 충돌쌍이 발견되는 등 안전성에 문제가 있어 사용하지 않을 것을 권고하고 있다[17]. 또한 변경 전 LPRNG의 시딩 과정은 그 원리가 알려지지 않았으며, [5]에서 확률과 정보이론을 바탕으로 분석하였으나 암호학적 안전성 증명을 제공하지 못했다.

새로운 LPRNG는 이 부분을 개선하고자 시딩 과정을 암호학적 안전성이 증명된 HKDF(Hash-based Key Derivation Function)[18]를 모사한 방법으로 변경하였다(소스 코드에 "HKDF-like construction"란 주석이 있음). 이를 근거로 파라미터를 대입하면 키에 대응하는 값은 입력 풀에 blake2s를 적용하여 출력한 256비트 해시값이고, 솔트(salt)에 대응하는 값은 CPU 타이머에서 읽은 64비트 값이라고 추정할 수 있다. 그러나 HKDF의 경우 솔트를 추출(extract) 단계에서만 사용하고, 확장(expand) 단계에서 사용하지 않는다. 또한 카운터와 관련된 파라미터도 존재하지 않아 HKDF의 명세를 정확하게 따른다고 보기는 어렵다. 따라서 새로운 LPRNG의 시딩 과정의 안전성을 명확하게 하기 위해 HKDF 모사 방식의 안전성을 증명할 필요가 있다.

3) 빠른 키 교체

변경 전 LPRNG가 엔트로피 카운터 값을 기준으로 시딩 여부를 결정한 것에 비하여 새로운 LPRN

G는 시딩 주기를 60초로 설정하였다. 이 기준은 crng_fast_key_erasure 함수를 호출할 때 적용한다. 이러한 변경은 하나의 키로 생성할 수 있는 난수의 양을 최소화하려는 시도로 보이지만 이론적 배경이 아직 공개되지 않아 암호학적 관점에서 어떤 장점이 있는지 불분명한 상태이다. 하지만 시스템 보안 관점에서 보면 하나의 키가 메모리에 상주하는 시간을 줄여 공격자가 메모리로부터 키를 유추해 내는 것을 어렵게 만들 수 있다.

4) 잡음원 다각화

변경 전 LPRNG는 부팅 초기 잡음원 부족에 의한 안전성 문제를 가지고 있었다[7-10]. 새로운 LPRNG는 기존 다섯 가지의 잡음원에 새로운 잡음원 다섯 가지를 추가하여 총 10가지의 잡음원을 수집한다. 부팅 초기 낮은 엔트로피 때문에 발생할 수 있는 안전성 문제를 해결하려는 시도로 볼 수 있다. 또한 시드 추출 기준(엔트로피 추정기의 값)을 192비트에서 256비트로 상향 조정하였다. 이는 암호학적 요구사항을 준수하기 위한 시도로 보인다. 예를 들어 국제 표준 난수발생기 명세[12]는 블록암호 기반 난수발생기를 사용할 때 보안 강도(security strength)별 준수해야 할 최소 엔트로피를 요구하고 있다. 만약 블록암호 기반 난수발생기가 256비트 시드(키)를 사용한다면 시드의 엔트로피는 256비트 이상이어야 함을 의미한다.

잡음원을 다각화하고 시드 추출 기준을 상향하였으나 관련된 분석 결과가 없어 그 효과를 확인하기 어려운 상황이다. 두 가지 요소를 고려하여 변경 전 LPRNG가 가진 부팅 초기 잡음원 부족 문제가 어느 정도 해결되었는지 확인할 필요가 있다.

### 3.2 성능 관점에서 구조 변화

1) 차단 모드 비활성화

차단 모드 비활성화는 낮은 엔트로피 카운터로 인해 발생한 난수 생성 지연 문제를 해결하기 위해 도입된 것으로 추정된다. 변경 전 LPRNG에 /dev/random 방식으로 난수를 요청할 때 엔트로피 카운터가 기준값을 넘지 못하면 난수 출력을 차단하였다. 이런 특징으로 인해 난수 요청이 많아지면 시스템의 성능 저하를 유발할 수 있다.

새로운 LPRNG는 이런 현상을 해결하기 위해 부팅 초기에만 엔트로피 추정기를 동작시킨다. 엔트로피 추정기의 값이 256비트(READY 상태)에 도달하면 차단 모드를 비활성화하고 엔트로피 추정기를 관리하지 않는다. 따라서 이후부터 /dev/random과 /dev/urandom의 동작은 같아진다. 이런 변화로 사용자는 새로운 LPRNG에 난수를 요청하면 지연 없이 난수를 획득할 수 있다.

2) CPU 코어별 자녀 CRNG 할당

CPU의 코어마다 자녀 CRNG를 할당한 것은 여러 코어가 난수발생기 자원을 점유하려 할 때 발생하는 성능 저하를 해결하기 위해 도입된 것으로 추정된다. 변경 전 LPRNG는 싱글 코어만 고려하여 설계되었다. 이 구조에서는 난수발생기를 점유한 코어가 자원을 반환해야 다른 코어가 난수발생기를 사용할 수 있다. 따라서 멀티 코어 환경에서는 난수 생성이 지연될 수 있다.

이 문제를 해결하기 위해 새로운 LPRNG는 난수 출력 부분을 멀티 코어 환경에 적합한 구조로 변경하였다. 코어마다 자녀 CRNG를 할당하여 각 코어는 자신에게만 할당된 자녀 CRNG에 접근하여 지연 없이 난수를 생성할 수 있다.

3) CPU 코어별 인터럽트 잡음원 전용 입력 풀 할당

인터럽트 잡음원은 CPU의 각 코어마다 발생한다. 코어별 자녀 CRNG를 할당한 것과 같이 인터럽트 잡음원 전용 입력 풀(fast pool)도 성능 향상을 위해 도입된 것으로 추정된다. 각 코어는 인터럽트를 자신의 전용 풀에 축적하고, 축적된 인터럽트 잡음원을 1초 주기로 입력 풀에 혼합한다. 모든 코어가 입력 풀에 직접 접근하는 것보다 효율적일 것으로 예상할 수 있다.

이번 장에서는 변경 전 LPRNG와 새로운 LPRNG의 구조를 암호학적 관점과 시스템 관점에서 비교하였다. 변경 전 LPRNG가 가진 안전성 문제를 암호학적 방식으로 해결하려는 시도를 확인할 수 있었다. 또한 난수를 효율적으로 생성하기 위한 성능 관점의 구조 변화도 확인할 수 있었다.

## IV. 결 론

본 논문은 최근 이루어진 리눅스 의사난수발생기의 대대적인 구조 변경을 분석하였다. 특히, 혼합 함수, 시드 추출, 난수 생성 과정에 집중하여 변경 전

리눅스 의사난수발생기와의 구조적인 차이점을 식별하였다. 그 결과로 암호학적 관점과 성능 관점의 개선 사항을 확인할 수 있었다. 그러나 대대적인 구조 변경에도 관련된 이론이나 실험적 근거는 불분명한 상태로 남아있어 새로운 리눅스 의사난수발생기의 추가적인 안전성 분석과 검증이 필요하다. 이런 관점에서 본 논문이 리눅스 의사난수발생기의 구조 변화에 대한 이해를 높이고, 추가적인 연구와 안전성 분석을 위한 토대로 활용되기를 기대한다.

## References


[1] B. Barak and S. Halevi, "A model and architecture for pseudo-random generation with applications to /dev/random", Proceedings of the 12th ACM conference on Computer and communications security, pp. 203-212, Nov. 2005.

[2] 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", Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 647-658, Nov. 2013.

[3] Z. Gutterman, B. Pinkas and T. Reinman, "Analysis of the linux random number generator", 2006 IEEE Symposium on Security and Privacy (S&P'06), pp. 371-385, May. 2006.

[4] F. Coichon, C. Lauradoux, G. Salagnac and T. Vuillemin, "Entropy transfer in the linux random number generator", RR-8060, INRIA, pp.26, hal-00738638, Sep. 2012.

[5] P. Lacharme, A. Röck, V. Strubel and M. Videau, "The linux pseudorandom number generator revisited", Cryptology ePrint Archive, Paper 2013/251, Jan. 2012.

[6] Y. Yeom and J-S, Kang, "Probability distribution for the linux entropy estimator", Discrete Applied Mathematics, Vol. 241, pp. 87-99, May. 2018

[7] N. Heininger, Z. Durumeric, E. Wustrov and J. A. Halderman, "Mining your ps and qs: detection of widesperad weak keys in network devices", 21st USENIX Security Symposium, pp. 205-220, Aug. 2012.

[8] S. H. Kim, D. Han and D. H. Lee, "Predictability of android openssl's pseudo random number generator", Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 659-668, Nov. 2013.

[9] D. Kaplan, S. Kedmi, R. Hay, A. Dayan, "Attacking the linux prng on android: weaknesses in seeding of entropic pools and low boot-time entropy", 8th USENIX Workshop on Offensive Technologies, Jul. 2014.

[10] T. Yoo, J-S. Kang and Y. Yeom, "Recoverable random numbers in an internet of things operating system", Entropy Vol. 19, pp. 113, Mar. 2017.

[11] BSI, "Documentation and analysis of the linux random number generator, version: 5.9", Jan. 2024.

[12] ISO, "ISO/IEC 18031: Information technology - security techniques. random bit generation", 2011.

[13] M. Matsumoto and Y. Kurita, "Twisted GFSR generators", ACM Transaction on Modeling and Computer Simulation, Vol. 2, Issue 3, pp. 179-194, Jul. 1992.

[14] M-J. Saarinen and J-P. Aumasson, "The blake2 cryptographic hash and message authentication code (MAC)", RFC 7693, Nov. 2015.

[15] J-P. Aumasson and D-J. Bernstein, "SipHash: a fast short-input PRF", In Proceedings of the International

Conference on Cryptology in India, pp. 489-508, Dec. 2012.
〔16〕 Y. Nir and A. Langley, "Chacha20 and poly1305 for ietf protocols", RFC 8439, Jun. 2018.
〔17〕 E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths", Special Publication (NIST SP) 800-131A Revision 2, National Institute of Standards and Technology, Mar. 2019.
〔18〕 H. Krawczyk, "HMAC-based extract-and-expand key derivation function (HKDF)", RFC 5869, May. 2010.
〔19〕 E. Barker, J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", Special Publication (NIST SP) 800-90A, National Institute of Standards and Technology, Jun. 2015.

## 〈저자소개〉

유 태 일 (Taeill Yoo) 정회원
2010년 2월: 국민대학교 수학과 이학사
2014년 9월: 국민대학교 수학과 이학석사
2019년 9월: 국민대학교 금융정보보안 이학박사
2016년 12월~현재: ETRI 부설연구소 선임연구원
〈관심분야〉 암호학, 정보보호

노 동 영 (Dongyoung Roh) 정회원
2004년 2월: KAIST 수학과 이학사
2011년 2월: KAIST 수리과학과 이학박사
2011년 2월~20112년 4월: 국가수리과학연구소 연구원
2012년 4월~현재: ETRI 부설연구소 책임연구원
〈관심분야〉 암호학, 정보보호