TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 13/15 pp370–385
DOI: 10.26599/TST.2022.9010005
Volume 28, Number 2, April 2023

LETRNG — A Lightweight and Efficient True Random Number Generator for GNU/Linux Systems

Yucong Chen, Fangfang Zhu, Yanshan Tian, Shuaixin Xu, Lihong Han, Qingguo Zhou*, and Nam Ling

Abstract: Unpredictable and irreproducible digital keys are required to modulate security-related information in secure communication systems. True random number generators (TRNGs) rather than pseudorandom number generators (PRNGs) are required for the highest level of security. TRNG is a significant component in the digital security realm for extracting unpredictable binary bitstreams. Presently, most TRNGs extract high-quality "noise" from unpredictable physical random phenomena. Thus, these applications must be equipped with external hardware for collecting entropy and converting them into a random digital sequence. This study introduces a lightweight and efficient true random number generator (LETRNG) that uses the inherent randomness of a central processing unit (CPU) and an operating system (OS) as the source of entropy. We then utilize a lightweight post-processing method based on XOR and fair coin operation to generate an unbiased random binary sequence. Evaluations based on two famous test suites (NIST and ENT) show that LETRNG is perfectly capable of generating high-quality random numbers suitable for various GNU/Linux systems.

Key words: GNU/Linux system; true random number generator; complex system; inherent randomness; non-determinism

1 Introduction

As the Cyber-Physical-Social-Systems (CPSS) have expanded into multiple sectors such as the Internet-of-Things (IoT), smart cities, healthcare, intelligent transportation, and so on, various GNU/Linux systems in CPSS keep increasing. An encrypted communication channel is an important approach for interconnecting those devices. Because cryptographic security feature is important in communication protection[1–7], generating high-quality random numbers is critical for cryptography algorithms. A true random number generator (TRNG) is required to generate sufficient true random sequences in a timely and effective manner to ensure the security of real-time communications. The entropy production for a TRNG is a challenging task, which is a key part of the security system.

However, many popular random entropy sources such as keyboard input, mouse movements[8], hard disk activities, and other hardware events[9] in each modern general-purpose computer system, are incompatible with most of the GNU/Linux systems used in CPSS[10, 11]. Meanwhile, widely used random entropy production methods in GNU/Linux systems are flawed, resulting in a predictable and lower-quality noise sequence. In addition, some flaws that are related to security issues go undetected. Most embedded GNU/Linux systems have limited resources, and are unable to connect to

• Yucong Chen, Yanshan Tian, Shuaixin Xu, Lihong Han, and Qingguo Zhou are with School of Information Science and Engineering, Lanzhou University, Lanzhou 730000, China. E-mail: chenyc18@lzu.edu.cn; tianysh12@lzu.edu.cn; xushx19@lzu.edu.cn; hanlh16@lzu.edu.cn; zhouqg@lzu.edu.cn.
• Yucong Chen and Fangfang Zhu are with Institute of Modern Physics, Chinese Academy of Sciences, Lanzhou 730000, China. E-mail: chenyc@impcas.ac.cn; zhuff@impcas.ac.cn.
• Nam Ling is with Department of Computer Science and Engineering, Santa Clara University, Santa Clara, CA 95053 USA. E-mail: nling@scu.edu.
∗ To whom correspondence should be addressed.
Manuscript received: 2021-12-15; revised: 2022-02-20; accepted: 2022-02-21

© The author(s) 2023. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

external hardware typically used for generating true random numbers[12]. Consequently, it is critical to design and implement a lightweight and efficient TRNG based on some inherent characteristics.

A random number generator (RNG) is a device or algorithm that generates a random number or sequence of symbols. Traditionally, it can be divided into TRNG and pseudo-random number generator (PRNG), as shown in Fig. 1. The TRNG, also known as a hardware RNG, generates random numbers through physical processes. These RNGs can usually produce some subtle and statistical "noise", and the "noise" source[13] on these devices can also be common physical phenomena, such as audio and video noise[14], radio noise, thermal noise, and quantum phenomena. These random physical processes are not only completely unpredictable in theory but also confirmed by physical experiments. TRNGs are usually made up of three components: an energy conversion unit, a signal amplification unit, and an Analog-to-Digital Conversion (ADC) unit. The energy conversion unit converts other forms of energy into electrical energy (electrical signal); the signal amplification unit amplifies the electrical signal to the level that can be detected by the ADC; and the ADC converts the acquired analog signal into a digital signal[15–17]. Alternatively, the PRNG is also known as a deterministic random bit generator. It is an algorithm that can generate an "approximate" random number sequence based on a small set of initial values.

The traditional random number generator classification method cannot account for many of the current research results of RNG because some existing research results[18, 19] show that the structure of TRNG does not necessarily include the three traditional elements (energy converter, amplifier, and ADC). Therefore, some international standardization organizations give the classification of RNGs in relevant standards according to the actual needs, such as ISO/IEC 18031:2011, which divides RNGs into two types, as shown in Fig. 2. The first type is deterministic RNG (traditional PRNG); the second type is non-deterministic RNG (TRNG), which is composed of two subtypes: physical RNG (traditional hardware RNG) and non-physical RNG. The non-physical RNG is based on non-deterministic events, such as system time, hard disk seeks time, user interaction events (mouse movement, keyboard tapping, touch screen operating, etc.).

To design TRNGs without the use of external peripherals, some algorithms use the unpredictable behavior of internal computer architecture as the source of randomness[20], such as hard disk speed variation[21]. This study introduces a new TRNG based on the inherent randomness of the modern computer system to produce high-quality random numbers. The method used is not only lightweight and effective but also independent of a specific platform. Therefore, it can be widely used in various GNU/Linux systems. A post-processing function based on XOR and fair coin operations is implemented[22–25] to ensure that the final output is unbiased and robust.

This study is organized as follows: Section 2 discusses the various implementations of RNGs based on the non-determinism of a central processing unit (CPU) and an operating system (OS) in the literature, while Section 3 introduces the inherent randomness and race condition (coin-tossing) model in modern computer systems. Section 4 provides a detailed explanation of the proposed system, followed by Section 5, which analyses the distribution of coin-tossing results and the security performance of the proposed system. Section 6 concludes the study with some final remarks.

2 Related Work

True random numbers based on the inherent non-determinism of complex systems have received little attention in the recent literature. The work we reference

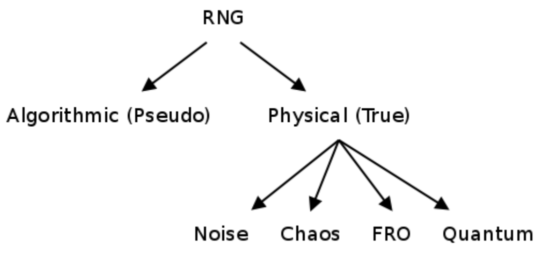


Fig. 1 Traditional classification of RNGs.

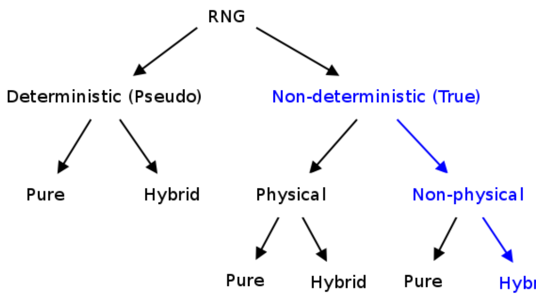


Fig. 2 Taxonomy of RNGs according to ISO/IEC 18031:2011.

here is not directly related though it does have similarities at the conceptual level.

Davis and Niphadkar[26] use the OS-level non-determinism (related to the mutex wakeup order) to build a multithreaded pseudo random number generator (MTPRNG). While the mutex wakeup mechanism is completely deterministic in terms of functionality, the order in which wakeup calls are executed is found to be non-deterministic to some extent if multiple threads of execution are waiting on a wakeup event simultaneously. This is known as a trampling-herd effect of mutexes with numerous threads sleeping on them. The MTPRNG library is implemented using 64 threads, each of which injects either 0 or 1 in a shared pool to generate a random 32-bit permutation. The results presented are inconclusive as to whether the entropy extracted using this method is suitable for cryptographic means.

AT&T's CryptoLib[27] is based on the randomness inherent in operating systems. The model is to have two physical timer sources and use the fact that they are never precisely synchronized to extract random bits from them. However, it is unclear if this is attributed to the physical non-determinism of the two clock sources or if it is attributed to the non-determinism of the OS when querying the two independent sources. The most valuable discovery is that complex systems (even by 1993 standards) exhibit inherent entropy and do not rely on external asynchronous events.

CPU time jitter-based non-physical true random number generator was brought up by S. Müller[28] and had been added to the mainline of the Linux kernel in the form of a patch in 2015. This is based on CPU execution time jitter and extracts entropy from different time delta while executing the same code snippet, then adds it to the kernel random device. The CPU jitter RNG reads a timestamp from a high-resolution timer to calculate a delta and folds it into one bit using a folding loop. Then, the value is processed with a von Neumann unbiased operation, which is added to the entropy pool using XOR. This operation is executed 64 times until all 64-bits of the entropy pool are filled with a new 64-bit random number. CPU jitter RNG gathers entropy using jent_get_nstime(), which is an architecture-dependent function with different implementations. In the Linux kernel space, the random_get_entropy() function obtains the high-resolution timestamp. In the user space, the clock_gettime() function is available for a high-resolution timer.

Mc Guire[29] proposed an embarrassingly simple random number generator (ESRNG) for GNU/Linux, which is based on the inherent non-determinism of the GNU/Linux. The principle of ESRNG is that when two writer threads execute concurrently to change the value of a shared variable, the results obtained by a reader thread each time are different, and it can be used as a source of entropy. The main advantage of ESRNG is that the method used to design such a software RNG is very simple. Furthermore, a histogram that records the intermediate values that occur in the inner loop should be added. Another advantage is that this is a pure software RNG that can run on almost any system that does not require any special hardware. ESRNG is based on trivial code and is implemented in user space as an unprivileged process. However, after collecting data extracted from the entropy source, ESRNG must eliminate "no-event" elements before putting the data into the entropy pool.

3 Preliminaries

In this section, we provide a short exhibition of inherent randomness and the coin-tossing model. Those concepts are used in the rest of this study.

3.1 Inherent randomness

RNGs are essentially based on some form of physical non-determinisms, such as a radioactive source[30]. The key point is that the physical phenomena of the component involved are unpredictable. The distribution of the phenomena used to extract entropy is an emerging property of RNG systems. Structural non-determinism is an emerging property of complex systems. Complex systems are often linked with non-deterministic chaos. Some systems exhibit complexity by virtue of being chaotic. However, a completely chaotic system is indistinguishable from one that behaves randomly[31]. The term structural non-determinism is used in other contexts, so we will refer to the concept outlined here as inherent randomness of the complex system[32–34].

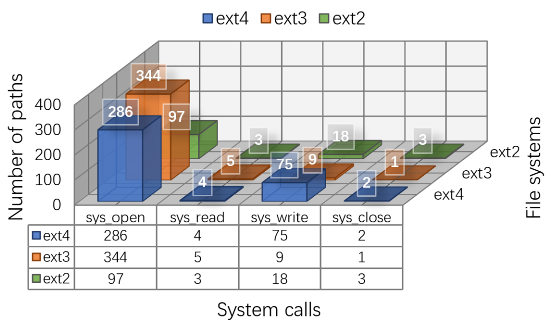
Currently, the sources of non-determinism or randomness in computer systems can be divided into two categories: hardware-related and software-related[35]. The hardware-related sources of randomness originate from CPU instruction pipelines (superscalar pipelining, multiple issue pipelining, and out-of-order execution), branch prediction units, CPU multi-level cache and memory hierarchy, the difference between CPU clock and memory bus clock, CPU performance scaling[36], CPU power management, hardware topology, or hardware interrupts resulting from external events. The software-related sources of randomness include randomized algorithms, randomized methods for preventing unauthorized access, and some complex components, such as memory allocator or task scheduler. In short, all types of non-determinism are inherent in current systems.

Among the many infrastructure software platforms, OS serves as the foundation for other variants of applications. It provides various services and interfaces, allowing users to implement a diverse set of applications. There are many popular open-source OSs such as GNU/Linux and FreeBSD, in which GNU/Linux is most widely used in our daily lives and workplaces, ranging from smartwatches to supercomputers. The Linux kernel is the innermost part of the GNU/Linux. We focus on the Linux kernel to demonstrate the non-determinism of current systems, which is a significant viewpoint for observing random phenomena.

3.1.1 Execution path diversity

The traditional idea about real-time systems is that the execution path and environment are determinism and predictability. However, in the Linux kernel, the system call execution path is highly unpredictable; it has a strong possibility because of the non-deterministic state of the execution environment[37–39]. The highly unpredictable feature of the system-call execution path in the Linux kernel is known as path variability, which could also have large differences between different system calls.

To illustrate the path diversity of a Linux kernel-based system, we used FTrace, a tracing utility built directly into the Linux kernel, to record the kernel space execution paths. The experiment was performed on a Raspberry Pi 3B. Besides, the hardware platform runs Ubuntu 20.04 with the kernel version 4.19.75 at idle. The objects of focus for tracing are some of the common system calls for file operations, such as sys_open, sys_read, sys_write, and sys_close. Figure 3 shows the number of execution paths in three file systems, Ext2, Ext3, and Ext4.



	sys_open	sys_read	sys_write	sys_close
ext4	286	4	75	2
ext3	344	5	9	1
ext2	97	3	18	3

Fig. 3 Number of paths for file operations in ext2, ext3, and ext4.

3.1.2 Execution time non-determinism

With the high complexity of modern CPUs and their operating systems, all the mentioned computer systems are extensively used. However, due to the hardware complexity, there are no suitable methods and techniques to determine which is the fill level of the caches or branch prediction units, or the precise location of data in memory at one given time[28]. We are also unable to predict when an interrupt/exception event will occur. This implies that the execution of function may have obvious variations in execution time.

To demonstrate the execution time distributions of the Linux kernel functions, we use Kprobes[40], a debugging mechanism that can also be used for monitoring events inside a production system, to monitor the execution time of the Linux kernel functions. The experiment is performed on a DE2i-150 FPGA development board, which is powered by an Intel Atom dual-core processor N2600 running at 1.6 GHz, and 2GB DDR3 SDRAM. Besides, the board runs the Linux Kernel version 4.19.75. Same as above, we still focus on the system calls related to the file systems. Figure 4 shows that the execution times distribution plot is spread out over such a large area, not several specific points.

3.1.3 Relationship among executive functions

Linux kernel consists of large quantities of functions, which collaborate through function calls and shared data structures. We also collected the function-call graphs from SIL4Linux[41] project to analyze the relationship among Linux kernel functions. The function call relationship uses functions as nodes and function calls as edges, as shown in Fig. 5, to illustrate the function call relationship among a set of Linux kernel functions, so the relationship can be represented as a directed network.

It is so difficult to describe the relationship among the kernel functions in a generic way, such as formulae, matrix, and simple graphs, that we have to use other different methods for analyzing the relationships, like McCabe's Cyclomatic Complexity[42] and complex network[43–45]. It is necessary to highlight that few studies have discussed the relationships among data structures in the Linux kernel. Meanwhile, no domestic scholar has conducted extensive research on the comprehensive combination of data structures and function calls.

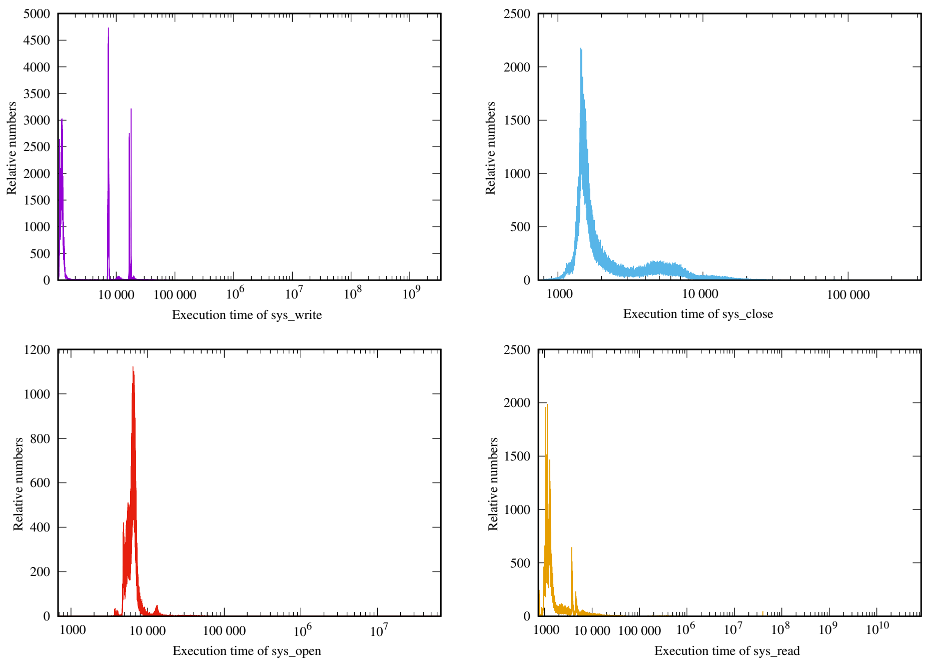


Fig. 4 Execution time distribution.

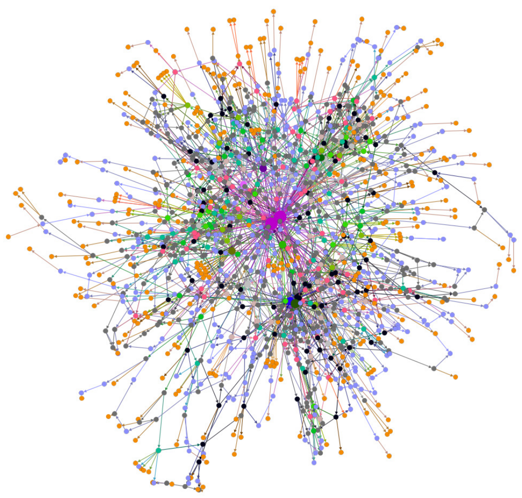


Fig. 5 Complex network in Linux kernel.

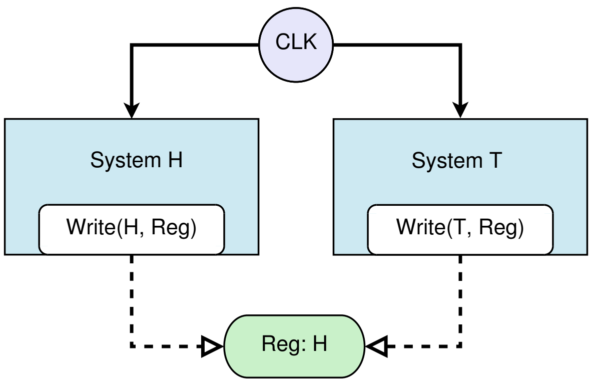


Fig. 6 Ideal deterministic race condition model.

3.2 Coin-tossing model

3.2.1 Deterministic race condition model

Figure 6 shows an ideal deterministic race condition model. The thought experiment goes as follows: take two systems of perfect deterministic design; assume that the systems are perfectly synchronous and jitter-free; add to these two systems a shared resource like a dual-ported register accessible to both systems with identical timing.

If both processors start at the same clock tick and execute the same code sequence, the difference is the values of Systems H and T writing to the shared register (H and T, respectively). Even though the entire setup is deterministic, the actual value found in the register is undetermined. The shared register cannot hold both H and T simultaneously. Thus, the perfectly deterministic hardware, paired with a deterministic execution sequence in software, produced a perfectly random result, $P(\text{H}) = P(\text{T}) = 1/2$. All that remains is to introduce points in the system where determinism is broken by a shared single state resource. This is the key concept behind a lightweight and efficient true random number generator (LETRNG). They harvest structural non-determinism rather than physical non-determinism.

The design problems of such a system are comparable to those of physical RNGs, bias problems, possibilities of the physical phenomena drifting, external influences such as thermal or voltage instability, and so on. Some hardware RNGs, such as the Intel RNG[46], use unbiasing methods to resolve bias issues, though this algorithm is only applicable if the bias is stable over time. Similarly, the structural non-determinism approach must convert potentially biased events into unbiased events and ensure stability in the presence of biased disturbances.

3.2.2 Race-based coin-tossing model

The remainder of this section proposes a lightweight race-based coin-tossing model to extract entropy for LETRNG. The objective is to provide a simpler, more efficient digital bitstream generator.

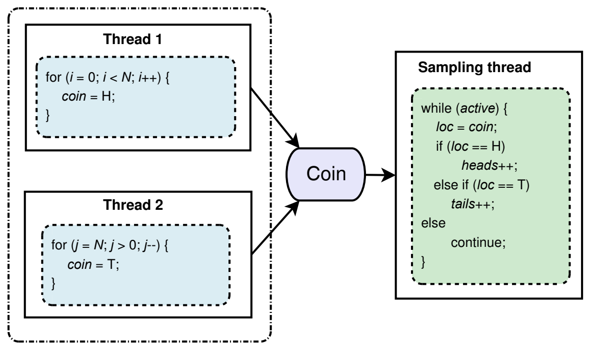
(1) Definitions and designs

Here we design a race condition intentionally, with the race being symmetric in the sense that we use two coin threads (Thread 1 and Thread 2) that modify the global variable multiple times by assignment operations, and one Sampling Thread that asynchronously sample the global variable *coin*. Asynchronous here means that the threads are independently scheduled. Independent means that they declare no explicitly shared data, except for the global variable; though they are threads belonging to the same process and operate in the same address space. Furthermore, they are not restricted by the OS scheduling them on a particular CPU or any other implicit coupling that the OS may introduce. While this is a relatively weak notion of independence, it is sufficient to provide a reliable source of entropy. The architecture of the coin-tossing model is shown in Fig. 7.

Thread 1 writes H to the *coin*, and Thread 2 writes T to the *coin*. Thus, a sequence of coin-tossing experiments is performed, with the Sampling Thread's observation sequence consisting of a series of heads and tails. A typical observation sequence would be H H T T T H T ⋯ T, where H stands for the head and T stands for the tail. The coin-tossing sampling method is shown in Algorithm 1. If the value read by Sampling Thread from the *coin* is H, then *heads* will be incremented by 1; if the value is T, *tails* will be incremented by 1.

(2) Probability computation

Suppose coin-tossing has $P(\text{heads}) = P(\text{H}) = p$, where



```
for (i = 0; i < N; i++) {
    coin = H;
}
```

```
for (j = N; j > 0; j--) {
    coin = T;
}
```

```
while (active) {
    loc = coin;
    if (loc == H)
        heads++;
    else if (loc == T)
        tails++;
    else
        continue;
}
```

Fig. 7 Coin-tossing model implementation.

Algorithm 1 Coin-tossing algorithm for sampler

Input: shared **unsigned int** *coin*, **boolean** *active*.
Output: **unsigned int** variables *heads* and *tails*.

```
wait writer thread
while $active == \text{True}$ do
    if $coin == \text{H}$ then
        $heads \leftarrow heads + 1$
    else if $coin == \text{T}$ then
        $tails \leftarrow tails + 1$
    else
        continue
return result
```

$p \in [0, 1]$, on each sampling. For $i \in \{1, 2, 3, ..., \text{N}\}$; let R_i = H or T according to *i*th coin-tossing sampling; define H = 1 and T = 0.

Let

$$S_n(\text{H}) = \sum_{i=0}^{n} R_i$$

be the number of heads in the *n* samplings.

Let

$$P_n(\text{H}) = \frac{S_n}{n}$$

be the proportion of heads in the *n* samplings.

The probability *p* that an unbiased coin-tossing result is H can be expressed as

$$p = P(\text{H}) = \lim_{n\to\infty} \frac{\sum_{i=0}^{n} R_i}{n} = \frac{1}{2}$$

as $n \to \infty$.

4 LETRNG System Design

This study proposed RNG which is based on the inherent randomness of OS and competitive conditions of complex computer systems. The core composition of the RNG is shown in Fig. 8.

LETRNG is composed of three parts: 1. Binary sequence generation component; 2. Binary data sampling component; 3. Entropy extraction and processing component. They are introduced below.

4.1 Binary sequence generation component

As shown in Fig. 8 and Algorithm 2, the binary sequence generation component is composed of two threads, named Thread 1 and Thread 2, which share a global variable *coin*. They write 0 or 1 into the shared variable via "*coin* = *n*% 2". There is no synchronization protection between Thread 1 and Thread 2, and they are running completely independently.

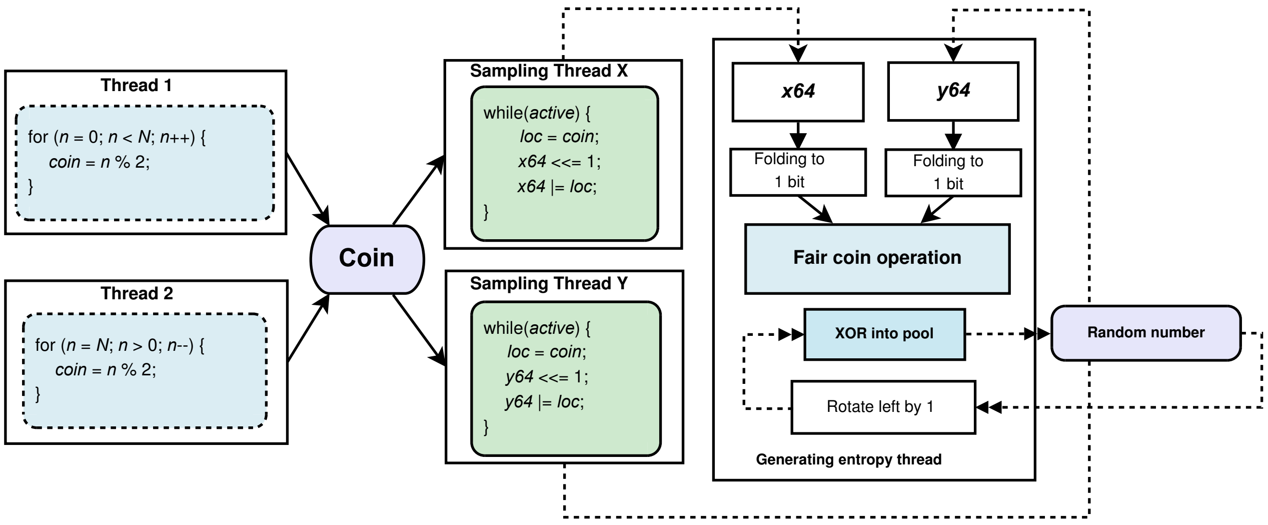


Fig. 8 Proposed LETRNG implementation.

Algorithm 2 Binary sequence generation algorithm

Input: number of iterations N.
Output: **boolean** *active*, shared **unsigned int** *coin*.

```
1: wait sampler thread
2: $active \leftarrow$ True
3: for each $n \in [1, 2, 3, \ldots, N]$ do
4:     $coin \leftarrow n\%2$
5: $active \leftarrow$ False
6: return result
```

Figure 8 shows that the cores of Thread 1 and Thread 2 are almost the same. The number of **for** loops is *N*. The only difference between them is as follows: Thread 1 executes an increment operation, whereas Thread 2 performs a decrement operation on the loop body count condition *n*.

Because the executions of Thread 1 and Thread 2 are not subject to any restrictions, they write 0 or 1 into shared variable *coin* simultaneously. Because of the statistics, the time for the shared variable *coin* holds the 0 or 1 should be equal. It is necessary to introduce a sampling thread to verify this inference. The *coin* read by the sampling thread is either 1 or 0, and the probability is 0.5.

The condition variable *active* was controlled by Thread 1 and Thread 2. When either of them completes **for** loop, the *active* will be set to false and the sampling thread is terminated. In addition, this study uses the volatile keyword to define shared variable *coin* to ensure that the sampling thread always reads data directly from the variable address.

4.2 Random binary sequence sampling component

According to the description in the previous section, Thread 1 and Thread 2 can write 0 and 1 to the global shared variable *coin* with equal probability without any protection of synchronization measures. The binary sequences generated by Thread 1 and Thread 2 must be sampled to obtain a random binary sequence. The sampling procedure described in the previous section is only used to experimentally verify the statistical distribution of 0 and 1 in the sampling results. However, the sampling procedure used in the RNG design process is different from that used for verification in the previous section.

Because the value of *n* of the core instruction "$coin = n\%2$" in Thread 1 and Thread 2 uses two different change strategies, which are increment and decrement, this study defines two variables *x64* and *y64* for the sampling procedure, and their data types are volatile unsigned long long, which is a 64-bit unsigned integer. The two sampling threads corresponding to *x64* and *y64* are Sampling Thread X and Sampling Thread Y, respectively. The two sampling threads save the binary sequence obtained from each sampling into *x64* and *y64*, respectively. The core function related to sampling in the sampling thread is shown in Algorithm 3.

Binary sequence generation and sampling threads must be executed concurrently in the RNG proposed in this study. However, it is necessary to use any synchronization mechanism between them, and they can run arbitrarily in the target system without restrictions.

Algorithm 3 Sampling random binary stream algorithm

```
Input: shared unsigned long long coin and loc, boolean active.
Output: unsigned long long variable x64.
1: sampler thread:
2: while $active == $ True do
3:     $loc \leftarrow coin$
4:     $x64 \leftarrow x64 \cap (\neg 0x1LL)$
5:     $x64 \leftarrow x64 \cup loc$
6:     $x64 \leftarrow x64 \ll 1$
7: return result
```

When the binary sequence generation threads execute the given number loops, it sets the condition variable *active* to false. Thus, if the sampling thread detects that the *active* is false, it will stop sampling.

Because the size of these two storage variables was determined when *x64* and *y64* were initially defined, the length of the binary sequence that the sampling thread can store will not exceed 64 bits, so the two Sampling Threads can only save the latest 64-bit data. The following three operations were also used: AND, OR, and SHIFT. In each sampling operation, the sampling thread first reads the global shared variable *coin* and saves it into the local variable *loc*, then cleared the lowest bit of *x64* (*y64*). Then, the collected data is saved to the lowest bit of *x64* using the OR operation. Finally, the overall arithmetic of *x64* shifted left 1-bit. In addition, because Thread 1, Thread 2, Sampling Thread X, and Sampling Thread Y shared the global variable *coin*, there were competitive relationships between them.

4.3 Post-processing sampling results

LETRNG follows similar post-processing methods as the CPU Jitter RNG[28]. LETRNG defined a 64-bit unsigned integer for caching random binary sequences. This integer is also known as an "entropy pool". This study specifically introduced a thread for generating entropy, known as Generating Entropy Thread, to effectively process the binary sequence collected by Sampling Thread X and Sampling Thread Y. This thread executes a loop body dedicated to processing binary sequences (collecting entropy). The loop body mainly includes the following key operations:

Step 1. Extract sampling results *x64* and *y64* from Sampling Thread X and Sampling Thread Y;

Step 2. Use XOR operation to combine the sampled 64-bit binary sequence into 1-bit, that is, *x64* and *y64* are combined into 1-bit;

Step 3. Use fair Coin Operation proposed by von Neumann to process the two bits generated in Step 2;

Step 4. Use XOR operation to add one random bit generated in Step 3 to the entropy pool;

Step 5. Shift the entropy pool to the left circularly to prepare storage space for the next random bit.

To fill the entropy pool, the above five steps must be performed 64 times. When the entropy pool is filled, a 64-bit random number can be provided to the caller (user). The following five key operations will be discussed in detail.

4.3.1 Extracting sampling results

The method used by LETRNG to extract the sampling results is very simple, consisting of a read operation on ordinary 64-bit unsigned integer variables. However, the extraction operation cannot be executed immediately at any time, because any extraction operation would be blocked before Sampling Thread X and Sampling Thread Y finish sampling (that is, every bit in *x64* and *y64* was updated). When the extraction operations were successful in Generating Entropy Thread, it would notify the sampling thread to start sampling.

4.3.2 Merging binary sequences

After the sampling results were extracted, the 64-bit *x64 (y64)* must be combined into one bit using an XOR operation. The specific method is shown in Algorithm 4.

The fold_bits function contained a **for** loop body, with a fixed number of loops of 64, which corresponded to the data types of *x64* and *y64*. In addition, in the process of merging data bits, a shift operation was used in the loop body in addition to the XOR operation.

4.3.3 Fair coin operation

According to Section 4.2 in RFC 4086[47], the fair coin operation proposed by John von Neumann can eliminate any potential tendency or bias in the coin-tossing test. In probability theory and statistics, a series of independent Bernoulli tests whose probability of each test is equal to 0.5 can be called the fair coin. If the probability of a test is not equal to 0.5, it can only be called a biased or unfair coin. In many theoretical studies, it is usually assumed that a fair coin is an ideal

Algorithm 4 Folding random binary stream algorithm

```
Input: unsigned long long $B_i$.
Output: unsigned long long $R_o$.
1: for each $n \in [1, 2, 3, \ldots, 64]$ do
2:     $T_i \leftarrow B_i \ll (64 - n)$
3:     $T_i \leftarrow T_i \gg 63$
4:     $R_o \leftarrow R_o \oplus T_i$
5: return result
```

coin. However, some coin-tossing experiments show that many "coins" in the real world have a certain tendency, which is not the ideal fairness[48].

The probability and statistical characteristics of coin-tossing experiments are used as examples in some elementary or advanced textbooks, which assume that coins are fair or ideal. For example, the idea of the "random walk" of the mathematical-statistical model is based on a fair coin. To use the "unfair coin" to get a fair result, John von Neumann gives the following method:

(1) Flip the coin twice;
(2) If both the results are the same, discard the two results and start from the first step again;
(3) If the two results are different, the result of the first toss is used and the result of the second toss is discarded.

The reason Algorithm 5 can produce fair results is that the probability of getting the head of the coin first and then the tail must be equal to the probability of getting the tail of the coin first and then the head. Assume that the probability of getting the heads of a coin is β and the probability of getting the tails is $(1-\beta)$, then the probability of getting the head first and then the tail is $\beta(1-\beta)$, and the probability of getting the tail first and then the head is $(1–\beta)\beta$ because the coin does not change its propensity in two coin tosses. However, the two coin tosses are completely independent. The above method only obtains the results of one experiment, and it does not change the tendency of subsequent experiments. By repeating the three steps of the aforementioned method, the events with the same results can be eliminated, so that only two results with the same probability are generated in the coin-tossing experiment. Note that the above method is effective only when the coin-tossing operation is performed in pairs, and the probability of a certain side of a coin cannot be 0.

Algorithm 5 Fair coin algorithm

```
Input: shared unsigned in $C_a$ and $C_b$.
Output: unsigned long long variable $V$.
1:  sampler thread:
2:  while True do
3:      $A \leftarrow$ toss($C_a$)
4:      $B \leftarrow$ toss($C_b$)
5:      if $A == B$ then
6:          continue
7:      if $A == 1$ then
8:          $V \leftarrow 1$
9:      else
10:         $V \leftarrow 0$
11: return result
```

From the above discussion, we must collect and process two random bits to perform a fair coin operation, which is the key reason we define two sampling threads (Sampling Thread X and Sampling Thread Y) and two integer variables (*x64* and *y64*) to cache 64-bit binary sequences.

4.3.4 Filling entropy pool

After the 64-bit sampling results are merged into 1-bit and the von Neumann fair coin operations, the final result must be added to the entropy pool. LETRNG will XOR the latest random bit with a bit that is in the entropy pool and store the result in the entropy pool. Therefore, which bit in the entropy pool is selected for XOR operation? LETRNG selects the lowest bit in the entropy pool and performs a circular left shift operation on the entire entropy pool, only circularly shifting 1-bit to the left each time because only 1-bit has to be added to the entropy pool each time after the fair coin operation.

5 Experiments

We select the following as our primary targets for evaluating LETRNG: (1) distribution of coin-tossing results, (2) quality of random numbers generated, and (3) the bandwidth of the proposed TRNG in each test platform.

5.1 Experimental setup

Even though we did run LETRNG on a relatively large number of platforms and have attempted to test as different systems as possible, this does not cover all possibilities. Notably, on low-end systems, we would expect that the current settings may be insufficient. Below is a summary of systems used in Table 1.

All systems could reach the 1 GB sample size, which is recommended for analysis with the NIST test suit. An unprivileged user executed the LETRNG instance as a user-space task. All systems are connected to LAN. All systems were idle but running in a multi-user mode; no special measures were taken to limit load or create any particular load scenario.

5.2 Coin-tossing results

After performing coin-tossing experiments on Intel E5-2650 v4 with Ubuntu 16.04 and ICT Loongson-3B V0.7 with CentOS 6.4, the results shown in Figs. 9 and 10. The distributions in the sampling results are consistent, and the probability of the heads and tails of the coin is

Table 1 List of tested system configurations.

CPU	GNU/Linux distribution	Kernel	State	RAM (GB)
Intel Core i7-4790	Debian 8.7	3.16.0	IDLE	8
Intel Core i7-4790K	Debian 8.6	3.16.0	IDLE	16
Intel Xeon E5-2620	Ubuntu 14.10	3.19.0	IDLE	8
Intel Xeon E5530	Debian 7.11	3.2.0	IDLE	6
Intel Xeon E5-2630	CentOS 7.2	3.10.0	IDLE	32
Jetson-TK1 ARMv7	Ubuntu 14.04	3.10.40	IDLE	2
Raspberry Pi 3B ARMv7	Debian 8.0 (32-bit)	4.4.11	IDLE	1
Raspberry Pi 3B ARMv8	Ubuntu 20.04 (64-bit)	5.4.0	IDLE	1
AMD Athlon X4 750	Debian 8.1	3.16.0	IDLE	4
Loongson-3A2000	Debian 8.5	3.16.7	IDLE	2
Jetson-TX1 ARMv8	Debian 8.3	3.10.67	IDLE	4
Intel Core i7-9700K	Debian 10.2	4.19.0	IDLE	64
Intel Core i3-8145UE	Ubuntu 18.04	5.4.0	IDLE	16
Xilinx XCZU3EG	Debian 10.11	5.4.0	IDLE	4
Raspberry Pi 4B ARMv7	Raspbian 10 (32-bit)	4.19.75	IDLE	4
Raspberry Pi 4B ARMv8	Ubuntu 20.04 (64-bit)	5.13.0	IDLE	4
Loongson-3A5000	Loongnix 20	4.19.0	IDLE	16
NanoPC-T4 RK3399	Ubuntu 20.04	5.10.60	IDLE	4
Intel Atom N2600	Ubuntu 21.04	5.11.0	IDLE	2

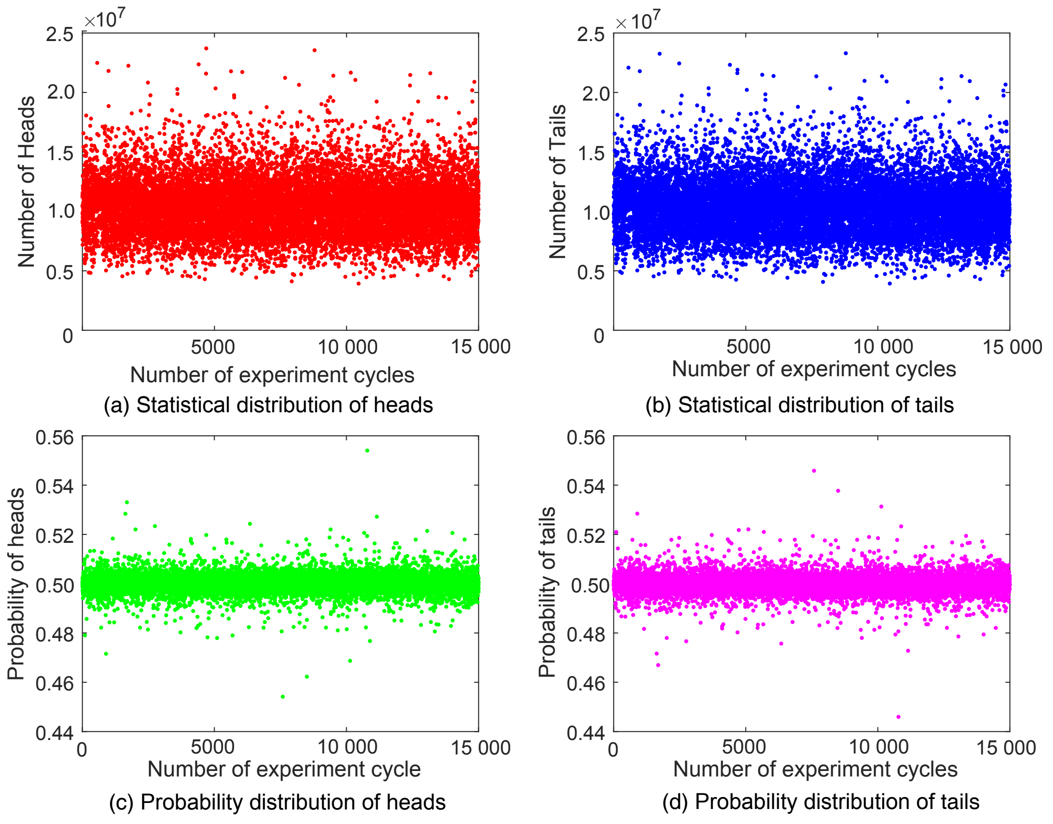


Fig. 9 Coin-toss sampling distribution of heads and tails on E5-2650 V4.

equal to 0.5. The above experimental results show that Thread 1 and Thread 2 of LETRNG can write sequences of **H** and **T** to the global shared variable *coin* with equal probability.

5.3 ENT and NIST test results

The security of LETRNG is evaluated using several criteria in this section. Firstly, the black-box statistical testing is executed to ensure that the output of

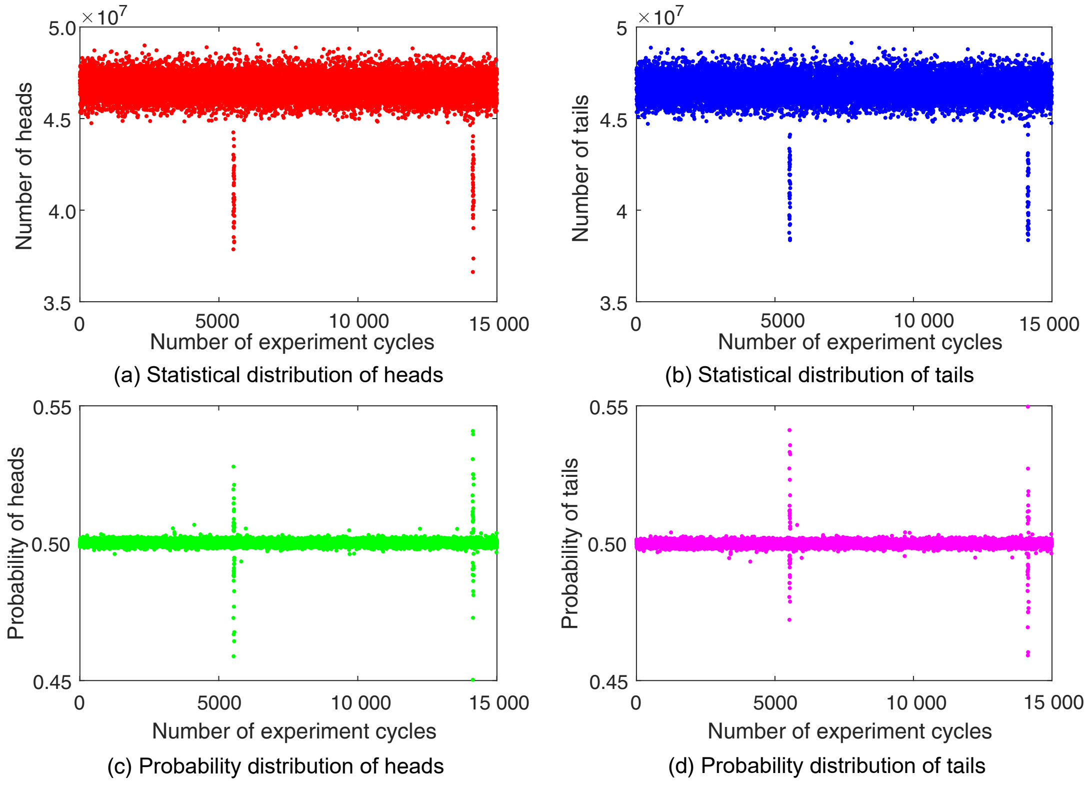


Fig. 10 Coin-toss sampling distribution of heads and tails on Loongson3B V0.7.

LETRNG has a uniform distribution and better statistical characteristics[49]. The procedure for testing consists of two steps. First, the production of the random sequences was monitored using the ENT suite[50] which is a collection of five statistic tools (Entropy, Chi-square test, Arithmetic mean, Monte Carlo value for Pi, and Serial correlation coefficient). ENT suite was used for screening to detect LETRNG failures, although it is not very exacting, it allows to discard weak designs that commonly fail the Chi-square test[51]. Secondly, we have tested the binary file with the NIST SP 800-22 Statistical Test Suite[52] to which we submitted 1024 bit-streams, each 1 MB in size, generated by the test systems, running all 15 tests. The parameters used for the NIST test are shown in Table 2.

Table 2 NIST STS-2.1.2 parameter settings used.

Parameter	Value
Significance level	0.01
Bitstream length	8 388 608
Block frequency block length	128
Nonoverlapping template block length	9
Overlapping template block length	10
Approximate entropy block length	10
Serial block length	16
Linear complexity sequence length	500

NIST SP 800-22 explains the calculation method based on Formula (1) of confidence intervals for statistics test items.

$$p' \pm 3\sqrt{\frac{p(1-p')}{n}} \tag{1}$$

where n is the number of samples and p' equal to $1-\alpha$, α is significance level. To ensure the accuracy of the test results, n must be greater than or equal to 1000.

Because the random bit-stream generated by LETRNG is stored in a file, we must first obtain those files generated on different platforms. In the experiment, the size of the random files generated on each platform is different, with the sample size ranging from 0.5 GB to 3.1 GB. The detail is shown in Tables 3 and 4, which also show the test results of each platform. STS-2.1.2 which is the latest version of the officially provided test software is used for the NIST test set. Furthermore, the parameters used throughout the test were those recommended in the documentation. In the testing process using NIST, we only test 1 GB data and divide it into 1024 bitstreams whose size is 1 MB for testing samples larger than 1 GB. The data are not tested if their sample size is less than 1 GB. According to the value calculated using Formula (1), the confidence interval of the tested samples is [0.99 − 0.009328022, 0.99 +

Table 3 ENT test results.

System	Sampling size (GB)	Entropy	Chi-square	Arithmetic mean	Monte Carlo value for Pi	Serial correlation coefficient
Intel Core i7-4790	1.730	8.000 000	236.32	127.4975	3.141 701 586	0.000 014
Intel Core i7-4790K	2.640	8.000 000	247.43	127.5004	3.141 629 920	0.000 016
Intel Xeon E5-2620	1.980	8.000 000	235.24	127.5005	3.141 519 611	−0.000 008
Intel Xeon E5530	1.960	8.000 000	272.43	127.5003	3.141 602 775	−0.000 013
Intel Xeon E5-2630	2.370	8.000 000	215.83	127.5012	3.141 658 791	−0.000 021
Jetson-TK1 ARMv7	2.000	8.000 000	260.74	127.4965	3.141 584 586	−0.000 033
Raspberrypi 3B ARMv7	0.776	8.000 000	257.83	127.4986	3.141 876 825	0.000 038
Raspberrypi 3B ARMv8	1.1	8.000 000	216.10	127.5006	3.141 734 010	0.000 006
AMD Athlon X4 750	0.496	8.000 000	258.95	127.4944	3.141 842 928	0.000 017
Loongson-3A2000	0.588	8.000 000	236.39	127.4984	3.141 724 119	−0.000 073
Jetson-TX1 ARMv8	0.738	8.000 000	231.66	127.4963	3.141 536 597	0.000 014
Intel Core i7-9700K	3.1	8.000 000	248.54	127.5014	3.141 566 541	0.000 000
Intel Core i3-8145UE	2.6	8.000 000	263.36	127.5001	3.141 503 002	−0.000 010
Xilinx XCZU3EG	1.4	8.000 000	265.21	127.5016	3.141 516 272	−0.000 021
NanoPC-T4 RK3399	1.1	8.000 000	239.28	127.4971	3.141 634 115	−0.000 028
Raspberry Pi 4B ARMv7	0.95	8.000 000	222.30	127.5052	3.141 532 996	0.000 018
Raspberry Pi 4B ARMv8	1.1	8.000 000	222.16	127.4983	3.141 466 001	−0.000 011
Loongson-3A5000	1.4	8.000 000	259.89	127.4972	3.141 591 849	−0.000 042
Intel Atom N2600	1.1	8.000 000	231.23	127.4986	3.141 616 041	−0.000 024

Table 4 NIST test results.

System	Sampling size (GB)	Number of testing bitstreams	Failed statistical test items	Result
Intel Core i7-4790	1.73	1024	null	Passed
Intel Core i7-4790K	2.64	1024	null	Passed
Intel Xeon E5-2620	1.98	1024	null	Passed
Intel Xeon E5530	1.96	1024	null	Passed
Intel Xeon E5-2630	2.37	1024	null	Passed
Jetson-TK1 ARMv7	2.00	1024	null	Passed
Raspberry Pi 3B ARMv7	1.44	1024	OverlappingTemplate	Passed
Raspberry Pi 3B ARMv8	1.1	1024	null	Passed
AMD Athlon X4 750	1.18	1024	null	Passed
Loongson-3A2000	1.25	1024	null	Passed
Jetson-TX1 ARMv8	1.93	1024	null	Passed
Intel Core i7-9700K	3.1	1024	null	Passed
Intel Core i3-8145UE	2.6	1024	null	Passed
NanoPC-T4 RK3399	1.3	1024	null	Passed
Xilinx XCZU3EG	1.4	1024	null	Passed
Raspberry Pi 4B ARMv7	1.1	1024	null	Passed
Raspberry Pi 4B ARMv8	1.1	1024	null	Passed
Loongson-3A5000	1.4	1024	null	Passed
Intel Atom N2600	1.1	1024	null	Passed

0.009328022] for 1 GB data. In other words, the confidence interval is roughly between [0.9806, 1] and at least 1004 samples of 1024 samples pass the test.

According to Table 3, the random bit stream files generated by LETRNG passed the five tests of ENT on all test platforms. Table 4 shows that all the platforms passed the tests except Raspberry Pi 3B ARMv7 because it has one item that did not pass the tests. According to the final analysis report of the platform, the p-value of the failed overlapping template test item is 0.22574, which is larger than the average $\alpha = 0.01$ and should be identified as a random value in theory. However, the number of passed samples is 1003, and the value calculated from the confidence interval is 1004. Furthermore, there may be abnormality conditions in the test process, and it is normal if an item does not pass the test. Therefore, we can obtain the conclusion that the RNG designed in this study runs on different platforms has passed all the tests of test suites, meeting the requirements of the encryption applications for random numbers.

5.4 Bandwidth analysis

Table 5 lists the bandwidth of random sequence generation in LETRNG of different hardware platforms. The table shows that the bandwidths in most platforms are relatively high, which have significant advantages compared with the bandwidth of random number generation by /dev/random with the method used in Linux. The bandwidth on embedded systems is relatively low because of the lower CPU frequency on these platforms. In the experiment, we found an intriguing phenomenon that the speed of generating random sequence by LETRNG on the platform with Intel Atom N2600 dual-core four threaded processor with the main frequency of 1.6 GHz is slower than those on platforms with Loongson-3A2000 quad-core processor with the main frequency of 0.8 GHz or BCM2837 quad-core processor with the main frequency of 1.2 GHz. Loongson-3A2000 belongs to MIPS architecture while Jetson-TX1, Jetson-TK1, Raspberry Pi, NanoPC-T4 RK3399, and Xilinx XCZU3EG are equipped with an ARM-base processor. MIPS and ARM are two different instruction set architectures in the family of Reduced Instruction Set Computing (RISC), while Intel Atom series processors are based on Complex Instruction Set Computers (CISC) architecture. Therefore, we suspect that the non-determinism of platforms based on RISC is more significant than CISC processors.

6 Conclusion

This study presented a lightweight and efficient TRNG based on the inherent non-determinism of the modern computer system. The proposed framework and algorithm take advantage of modern complex CPU and OS caused by unpredictabilities CPU jitter, race condition, and so on. The proposed method was analyzed using common and effective statistical test suites NIST SP 800-22 and ENT. LETRNG passed all test items in these suites indicating its capability to generate high-quality random numbers. Meanwhile, LETRNG has a throughput of more than 3.5 kbps that fully satisfies the requirements for GNU/Linux devices in most of the typical applications of CPSS. In conclusion, LETRNG can generate true random numbers appropriate for security-related applications at a sufficient bandwidth without requiring any extra hardware.

Acknowledgment

The authors would like to thank Nicholas Mc Guire for his thoughtful advices and review. This work was partially supported by National Key R&D Program of China (No. 2020YFC0832500), Fundamental Research Funds for the Central Universities (Nos. lzujbky-2021-sp47, lzujbky-2020-sp02, lzujbky-2019-kb51, and lzujbky-2018-k12), and the National Natural Science Foundation of China (No. 61402210). We also gratefully acknowledge the support of NVIDIA Corporation with the donation of the Jetson-TX1 used for this research.

References

[1] B. Kerrigan and Y. Chen, A study of entropy sources in cloud computers: Random number generation on cloud hosts, in *Proc. 6th Int. Conf. on Mathematical Methods, Models and Architectures for Computer Network Security*, St. Petersburg, Russia, 2012, pp. 286–298.

[2] F. Goichon, C. Lauradoux, G. Salagnac, and T. Vuillemin, Entropy transfers in the Linux random number generator, HAL preprint HAL Id: 1409.4842, 2014.

[3] L. Sumter, Cloud computing: Security risk, in *Proc. 48th Ann. Southeast Regional Conf.*, Oxford, MS, USA, 2010, p. 112.

[4] M. Mowbray and S. Pearson, A client-based privacy manager for cloud computing, in *Proc. 4th Int. ICST Conf. on communication System software and middleware*, Dublin, Ireland, 2009, p. 5.

[5] D. Lin and A. Squicciarini, Data protection models for service provisioning in the cloud. in *Proc. 15th ACM Symp.*

Table 5 Bandwidth of LETRNG on test systems.

System	Throughout (kbps)	System	Throughout (kbps)	System	Throughout (kbps)
Intel Core i7-4790	69.53	Intel Core i7-4790K	74.24	Intel Xeon E5-2620	57.00
Jetson-TK1 ARMv7	22.88	Intel Xeon E5530	82.72	Raspberry Pi 3B ARMv7	5.80
Raspberry Pi 3B ARMv8	18.51	Intel Xeon E5-2630	75.20	AMD Athlon X4 750	13.86
Jetson-TX1 ARMv8	16.48	Loongson-3A2000	5.70	Intel Core i7-9700K	87.00
Intel Core i3-8145UE	15.97	NanoPC-T4 RK3399	68.45	Xilinx XCZU3EG	15.62
Raspberry Pi 4B ARMv7	12.15	Raspberry Pi 4B ARMv8	12.80	Loongson-3A5000	19.62
Intel Atom N2600	3.56				

on Access Control Models and Technologies, Pittsburgh, PA, USA, 2010, pp. 183–192.

[6] C. Yuan, Y. Zhong, and S. Yang, Composite chaotic pseudo-random sequence encryption algorithm for compressed video, *Tsinghua Science and Technology*, vol. 9, no. 2, pp. 234–241, 2004.

[7] G. Zhong, K. Xiong, Z. Zhong, and B. Ai, Internet of things for high-speed railways, *Intelligent and Converged Networks*, vol. 2, no. 2, pp. 115–132, 2021.

[8] X. Wang, X. Qin, and L. Teng, A novel true random number generator based on mouse movement and a one-dimensional chaotic map, *Mathem. Probl. Eng.*, vol. 2012, p. 931802, 2012.

[9] K. Wallace, K. Moran, E. Novak, G. Zhou, and K. Sun, Toward sensor-based random number generation for mobile and IoT devices, *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1189–1201, 2016.

[10] Z. Gutterman, B. Pinkas, and T. Reinman, Analysis of the Linux random number generator, in *Proc. 2006 IEEE Symp. on Security and Privacy*, Berkeley/Oakland, CA, USA, 2006, pp. 371–385.

[11] T. Suzuki and M. Kaminaga, A true random number generator method embedded in wireless communication systems, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E103.A, no. 4, pp. 686–694, 2020.

[12] V. Fischer and M. Drutarovský, True random number generator embedded in reconfigurable hardware, in *Proc. 4^{th} Int. Workshop on Cryptographic Hardware and Embedded Systems*, Redwood Shores, CA, USA, 2002, pp. 415–430.

[13] M. Bucci and R. Luzzi, Design of testable random bit generators, in *Proc. 7^{th} Int. Workshop on Cryptographic Hardware and Embedded Systems*, Edinburgh, UK, 2005, pp. 147–156.

[14] I. T. Chen, Random numbers generated from audio and video sources. *Mathem. Probl. Eng.*, vol. 2013, p. 285373, 2013.

[15] S. Poli, S. Callegari, R. Rovatti, and G. Setti, Post-processing of data generated by a chaotic pipelined ADC for the robust generation of perfectly random bitstreams, in *Proc. 2004 Int. Symp. on Circuits and Systems*, Vancouver, Canada, 2004, p. IV–585.

[16] Y. Ma, T. Chen, J. Lin, J. Yang, and J. Jing, Entropy estimation for ADC sampling-based true random number generators, *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 11, pp. 2887–2900, 2019.

[17] E. Fatemi-Behbahani, K. Ansari-Asl, and E. Farshidi, A new approach to analysis and design of chaos-based random number generators using algorithmic converter, *Circuits Syst. Signal Process.*, vol. 35, no. 11, pp. 3830–3846, 2016.

[18] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, and D. Wichs, Security analysis of pseudo-random number generators with input: /dev/random is not robust. in *Proc. 2013 ACM SIGSAC Conf. on Computer & Communications Security*, Berlin, Germany, 2013, pp. 647–658.

[19] A. Colesa, R. Tudoran, and S. Banescu, Software random number generation based on race conditions, in *Proc. 2008 10^{th} Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, 2008, pp. 439–444.

[20] G. Souaki and K. Halim, Random number generation based on MCU sources for IoT application. in *Proc. 2017 Int. Conf. on Advanced Technologies for Signal and Image Processing* (*ATSIP*), Fez, Morocco, 2017, pp. 1–6.

[21] D. Davis, R. Ihaka, and P. Fenstermacher, Cryptographic randomness from air turbulence in disk drives, in *Proc. 14^{th} Annu. Int. Cryptology Conf. on Advances in Cryptology*, Santa Barbara, CA, USA, 1994, pp. 114–120.

[22] P. Lacharme, Post-processing functions for a biased physical random number generator, in *Proc. 15^{th} Int. Workshop on Fast Software Encryption*, Lausanne, Switzerland, 2008, pp. 334–342.

[23] J. S. Teh, W. Teng, A. Samsudin, and J. Chen, A post-processing method for true random number generators based on hyperchaos with applications in audio-based generators, *Front. Comput. Sci.*, vol. 14, no. 6, p. 146405, 2020.

[24] V. Rožić and I. Verbauwhede, Hardware-efficient post-processing architectures for true random number generators, *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 66, no. 7, pp. 1242–1246, 2019.

[25] L. Gantel, A. Duc, L. Steiner, F. Vannel, A. Upegui, and F. Gluck, A FPGA-based post-processing and validation platform for random number generators, in *Proc. 2020 IEEE Int. Parallel and Distributed Processing Symp. Workshops*, New Orleans, LA, USA, 2020, pp. 123–126.

[26] M. Davis and S. Niphadkar, LibMTPRNG: A multithreaded pseudo random number generator, https://www.drdobbs.com/parallel/libmtprng-a-multithreaded-pseudo-random/216900024.

[27] J. B. Lacy, Cryptolib: Cryptography in software, in *Proc. 4^{th} USENIX UNIX Security Symp.*, Santa Clara, CA, USA, 1993.

[28] S. Müller, CPU time jitter based non-physical true random number generator, https://www.chronox.de/jent/doc/CPU-Jitter-NPTRNG.html, 2013.

[29] N. Mc Guire, Principles and implementation of esrngs – embarrassingly simple random number generators for gnu/Linux, presented at the (4th Real-Time Linex Workshop, Chapel Hill, NC, USA, 2012, p. 26.

[30] A. Alkassar, T. Nicolay, and M. Rohe. Obtaining true-random binary numbers from a weak radioactive source. in *Proc. Int. Conf. on Computational Science and its Applications*, Singapore, 2005, pp. 634–646.

[31] J. Ladyman, J. Lambert, and K. Wiesner, What is a complex system? *Eur. J. Phil. Sci.*, vol. 3, no. 1, pp. 33–67, 2013.

[32] N. Mc Guire, P. Okech, and G. Schiesser, Analysis of inherent randomness of the Linux kernel, presented at the 11th Real-Time Linux Workshop, Dresden, Germany, 2009, p. 41.

[33] V. M. Weaver, D. Terpstra, and S. Moore, Non-determinism and overcount on modern hardware performance counter implementations, in *Proc. 2013 IEEE Int. Symp. on Performance Analysis of Systems and Software*, Austin, TX, USA, 2013, pp. 215–224.

[34] J. Hughes and J. O'Donnell, Expressing and reasoning

about non-deterministic functional programs, in *Proc. 1989 Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, 1989, pp. 308–328.

[35] M. Hocko and T. Kalibera. Reducing performance non-determinism via cache-aware page allocation strategies. in *Proc. 1st Joint WOSP/SIPEW Int. Conf. on Performance Engineering*, San Jose, CA, USA, 2010, pp. 223–234.

[36] R. J. Wysocki, CPU performance scaling, https://www.kernel.org/doc/html/v5.4/admin-guide/pm/cpufreq.html, 2017.

[37] B. A. Nejmeh, NPATH: A measure of execution path complexity and its applications, *Commun. ACM*, vol. 31, no. 2, pp. 188–200, 1988.

[38] P. Okech, N. M. Guire, and W. Okelo-Odongo, Inherent diversity in replicated architectures, arXiv preprint arXiv: 1510.02086, 2015.

[39] P. Okech, N. McGuire, and C. Fetzer, Investigating execution path non-determinism in the Linux kernel, presented at 15th Real Time Linux Workshop, Lugano-Manno, Switzerland, 2013, p. 15.

[40] S. Goswami, An introduction to kprobes, https://lwn.net/Articles/132196/, 2005.

[41] L. Wang, C. Zhang, Z. Wu, N. Mc Guire, and Q. Zhou, SIL4Linux: An attempt to explore Linux satisfying sil4 in some restrictive conditions, presented at 11th Real-Time Linux Workshop, Dresden, Germany, 2009, p. 28.

[42] A. Jbara, A. Matan, and D. G. Feitelson, High-MCC functions in the Linux kernel, *Emp. Softw. Eng.*, vol. 19, no. 5, pp. 1261–1298, 2014.

[43] Y. Gao, Z. Zheng, and F. Qin, Analysis of Linux kernel as a complex network, *Chaos Solitons Fractals*, vol. 69, pp. 246–252, 2014.

[44] L. Wang, P. Yu, Z. Wang, C. Yang, and Q. Ye, On the evolution of Linux kernels: A complex network perspective, *J. Softw. Evol. Process.*, vol. 25, no. 5, pp. 439–458, 2013.

[45] L. Han, Q. Zhou, J. Zhang, X. Yang, R. Zhou, and J. Tang, Polymorphism and consistency: Complex network based on execution trace of system calls in Linux kernels, *Int. J. Mod. Phys. C*, vol. 31, no. 9, p. 2050126, 2020.

[46] G. Cox, C. Dike, and D. J. Johnston, Intel's digital random number generator (DRNG). In *Proc. 2011 IEEE Hot Chips 23 Symp.* (*HCS*), Stanford, CA, USA, 2011, pp. 1–13.

[47] D. E. Eastlake, J. I. Schiller, and Steve Crocker, *Randomness Requirements for Security*, https://doi.org/10.17487/RFC4086, 2005.

[48] P. Diaconis, S. Holmes, and R. Montgomery, Dynamical bias in the coin toss, *SIAM Rev.*, vol. 49, no. 2, pp. 211–235, 2007.

[49] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, et al., *Sp 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, Gaithersburg, MD, USA: National Institute of Standards & Technology, 2010.

[50] J. Walker, ENT: A pseudorandom number sequence test program, https://www.fourmilab.ch/random/, 2008.

[51] C. Camara, H. Martín, P. Peris-Lopez, and L. Entrena, A true random number generator based on gait data for the internet of you, *IEEE Access*, no. 8, pp. 71642–71651, 2020.

[52] A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, and N. A. Heckert, A statistical test suite for random and pseudorandom number generators for cryptographic applications, https://doi.org/10.6028/NIST.SP.800-22R1A, 2010.

Yucong Chen received the BS degree in software engineering from Hunan University. He is currently a PhD candidate at Lanzhou University with his major research interests focusing on safety-critical systems, embedded systems, and real-time systems. Since 2015, he works as an engineer at the Institute of Modern Physics, Chinese Academy of Sciences, where he has participated in different research and engineering projects involving safety-related embedded systems.

Fangfang Zhu received the MS degree in computer architecture from Lanzhou University. She is currently an engineer at the Institute of Modern Physics, Chinese Academy of Sciences, where she has participated in different research and engineering projects involving open-source software for experimental physics and industrial control systems. Her research interest includes embedded systems and magnet power supply control systems for particle accelerators.

Shuaixin Xu received the BS degree in internet of things engineering, from Wuhan University of Technology, China in 2019. He is now a master student in the School of Information Science & Engineering, Lanzhou University. His research interests include system virtualization and operating system.

Lihong Han received the MS degree in 2009 from Lanzhou Jiaotong University, and earned the PhD degree in 2021 from Lanzhou University. He is currently an associate professor at the School of Statistics, Lanzhou University of Finance and Economics, Lanzhou, China, working in the fields of complex systems, data mining, and intelligent optimization.

Yanshan Tian received the BS degree from Southwest Normal University and the MS degree from the Beifang University of Nationalities, China. He is a PhD candidate in technology of computer application at the School of Information Science & Engineering, Lanzhou University. He is a vice professor with Ningxia Normal University in computer science. His research interests include high-performance computing with GPU, embedded and distributed systems, and random number generators.

Qingguo Zhou received the BS and MS degrees in physics from Lanzhou University in 1996 and 2001, respectively, and received the PhD degree in theoretical physics from Lanzhou University in 2005. Now he is a professor of Lanzhou University and working in the School of Information Science and Engineering. He is also a fellow of IET. He was a recipient of IBM Real-Time Innovation Award in 2007, a recipient of Google Faculty Award in 2011, and a recipient of Google Faculty Research Award in 2012. His research interests include safety-critical systems, embedded systems, real-time systems, and autonomous driving.

Nam Ling received the BEng degree from the National University of Singapore and the MS and PhD degrees from the University of Louisiana, Lafayette, USA. He is currently the Wilmot J. Nicholson Family Chair Professor (Endowed Chair) of Santa Clara University (USA) and the Chair of its Department of Computer Science & Engineering (since 2010). From 2010 to 2020, he was the Sanfilippo Family Chair Professor (University Endowed Chair) of Santa Clara University. From 2002 to 2010, he was an associate dean for its School of Engineering. He is/was also a Cuiying Chair Professor for Lanzhou University, a guest professor for Tianjin University, a chair professor and Minjiang Scholar for Fuzhou University, a distinguished professor for Xi'an University of Posts & Telecommunications, a guest professor for Shanghai Jiao Tong University, a guest professor for Zhongyuan University of Technology (China), and a consulting professor for the National University of Singapore. He has more than 220 publications (including books) in video/image coding and systolic arrays. He also has seven adopted standards contributions and has been granted more than 17 U.S. patents. He is an IEEE Fellow due to his contributions to video coding algorithms and architectures. He is also an IET Fellow. He was named IEEE Distinguished Lecturer twice and was also an APSIPA Distinguished Lecturer. He received the IEEE ICCE Best Paper Award (First Place) and the IEEE Umedia Best/Excellent Paper Awards (three times). He received six awards from Santa Clara University, four at the University level (Outstanding Achievement, Recent Achievement in Scholarship, Presidents Recognition, and Sustained Excellence in Scholarship), and two at the School/College level (Researcher of the Year and Teaching Excellence). He has served as Keynote Speaker for IEEE APCCAS, VCVP (twice), JCPC, IEEE ICAST, IEEE ICIEA, IET FC & U-Media, IEEE U-Media, Workshop at XUPT (twice), ICCIT, as well as a Distinguished Speaker for IEEE ICIEA. He is/was General Chair/Co-Chair for IEEE Hot Chips, VCVP (twice), IEEE ICME, IEEE U-Media (five times), and IEEE SiPS. He was an Honorary Co-Chair for IEEE Umedia. He has also served as Technical Program Co-Chair for IEEE ISCAS, APSIPA ASC, IEEE APCCAS, IEEE SiPS (twice), DCV, and IEEE VCIP. He was Technical Committee Chair for IEEE CASCOM TC and IEEE TCMM, and has served as Guest Editor/Associate Editor for *IEEE TCAS-I*, *IEEE J-STSP*, Springer *JSPS*, Springer *MSSP*, and other journals. He has delivered more than 120 invited colloquia worldwide and has served as Visiting Professor/Consultant/Scientist for many institutions/companies.