

On Random Number Generation for Kernel Applications

Kunal Abhishek*

Society for Electronic Transactions and Security (SETS)

Chennai, India

kunalabh@gmail.com

E. George Dharma Prakash Raj

School of Computer Sciences, Engineering and Applications

Bharathidasan University, Tiruchirappalli, India

georgeprakashraj@yahoo.com

Abstract. An operating system kernel uses cryptographically secure pseudorandom number generator (CSPRNG) for creating address space layout randomization (ASLR) offsets to protect memory addresses of processes from exploitation, storing users' passwords securely and creating cryptographic keys. However, at present, popular kernel CSPRNGs such as Yarrow, Fortuna and `/dev/(u)random` which are used by MacOS/iOS/FreeBSD, Windows and Linux/Android kernels respectively lack the very crucial property of non-reproducibility of their generated bitstreams which is used to nullify the scope of predicting the bitstream. This paper proposes a CSPRNG called Cryptographically Secure Pseudorandom Number Generator for Kernel Applications (KCS-PRNG) which generates non-reproducible bitstreams. The proposed KCS-PRNG presents an efficient design uniquely configured with two new non-standard and verified elliptic curves and clock-controlled Linear Feedback Shift Registers (LFSRs) and a novel method to consistently generate non-reproducible random bitstreams of arbitrary lengths. The generated bitstreams are statistically indistinguishable from true random bitstreams and provably secure, resilient to important attacks, exhibits backward and forward secrecy, exhibits exponential linear complexity, large period and huge key space.

Keywords: Random Number Generator, CSPRNG, LFSR, Elliptic Curve, Kernel Applications

*Address of correspondence: Society for Electronic Transactions and Security (SETS), MGR Knowledge City, C.I.T. Campus, Taramani, Chennai, 600113 - India.

1. Introduction

A random number generator (RNG) is classified in two basic classes [1]: first, a deterministic random number generator (DRNG) or a pseudorandom number generator (PRNG) which needs a seed value as input and produces random looking bitstreams using some deterministic algorithm. Second, a true random number generator (TRNG) which uses physical and non-physical sources to generate true randomness. It is imperative to note that unlike PRNG or DRNG, TRNG does not need any seed but uses non-deterministic effects or physical experiments to generate the true random bits [1]. The significant differences between TRNG and PRNG are that TRNG generates non-reproducible arbitrary length random bitstreams without using any seed or initializer whereas the PRNG generates arbitrary length pseudorandom bitstreams using a seed value or initializer. TRNG is slow, having infinite period, costly in deployment and has the possibility of manipulation. Unlike TRNG, PRNG has less development and deployment cost (no need of dedicated hardware) but can produce reasonably good random looking bitstreams.

The design goals of RNG heavily depend on its target applications. A simple application like stochastic simulations or Monte Carlo integrations may require RNG to generate nothing more than a random looking bitstream [1]. However, a sensitive application of RNG like an operating system kernel on top of which entire critical systems or applications run, certainly requires RNG to generate high quality pseudorandom bitstreams which are also provably secure, unpredictable and must be non-reproducible.

A kernel uses a RNG to create ASLR offsets [2], generate salts to securely store users passwords [3] and generate random keys to implement various cryptographic primitives such as authentication etc. The ASLR is one of the most important techniques used by the kernel (in special cases termed as Kernel-ASLR or KASLR) which randomizes the process layout to protect the locations of the targeted functions such as stack, heap, executable, dynamic linker/loader etc. [2]. The ASLR not only demands statistically qualified high quality pseudorandom number generator but also requires the output bitstream to be provably secure and unpredictable. Hence, a CSPRNG (or simply a PRNG with regular entropy inputs for unpredictability) is a preferred type of RNG for kernel applications. There are many good CSPRNGs which are implemented in various operating systems and are used by their kernels. Fortuna, Yarrow and /dev/(u)random are the popular CSPRNGs which are currently implemented by Windows, MacOS/iOS/FreeBSD and Linux/Android operating systems respectively [4, 5]. In this paper, a new CSPRNG which exhibits ‘non-reproducibility’ property of a TRNG is proposed taking security of the above kernel applications into consideration.

In particular, the key contributions of this paper are as follows:

- A novel CSPRNG design comprises of two non-standard and verified secure elliptic curves and nine LFSRs uniquely configured in a clock-controlled fashion to attain exponential linear complexity is used to construct the proposed KCS-PRNG.
- A novel architecture of the KCS-PRNG is proposed to mitigate the gap of ‘non-reproducibility’ property.
- Two new non-standard and verified elliptic curves are introduced in this paper which are used by the proposed KCS-PRNG to mitigate the gap of ‘non-reproducibility’ property. Both

elliptic curves are generated randomly over 256-bit prime fields to ensure cryptographic and implementation security.

- Extensive security analysis of the proposed KCS-PRNG is carried out to ensure theoretical security.
- Experimental validation and demonstration of statistical qualities of randomness using National Institute of Standards and Technology (NIST), Diehard, TestU01 test suites.
- Experimental validation and demonstration of ‘non-reproducibility’ property of the proposed KCS-PRNG.
- The proposed KCS-PRNG is compared with present kernel CSPRNGs such as Fortuna, Yarrow and dev/random and an existing PRNG [6]. The KCS-PRNG is also compared with an existing TRNG [7] in context of non-reproducibility of the generated random bitstreams.

Rest of the paper is organized as follows: Section 2 briefly discusses the randomness requirements of the kernel applications and standard RNG requirements. Section 3 reviews current CSPRNGs implemented by popular operating system kernels. Section 4 presents the proposed design of the KCS-PRNG. Subsequently, Section 5 presents the security analysis and Section 6 elaborates experimental validation and demonstration of the proposed KCS-PRNG. Section 7 presents the details of the two new elliptic curves selected over large prime fields for use in the proposed KCS-PRNG. Section 8 shows the important obtained results of the proposed KCS-PRNG. Section 9 briefly analyses the performance of KCS-PRNG. Section 10 compares KCS-PRNG with existing kernel CSPRNGs as well as recent PRNG, CSPRNG and TRNG used by various cryptographic applications. Finally, Section 11 concludes the findings of this paper.

2. Preliminaries

2.1. Randomness for Kernel Applications

One of the most important kernel applications that requires high quality randomness is ASLR [2] which is an efficient mitigation technique against remote code execution attacks by randomizing the memory address of processes to disable memory exploitation. The ASLR currently uses CSPRNG to randomize the logical elements contained in the memory objects at the time of pre-linking (at the time of installation of the application), per-boot (on every time the system boots), per-exec (when new executable image is loaded in memory called pre-process randomization), per-fork (every time a new process is created) and per-object (every time a new object is created). Figure 1 shows the Per-boot versus Per-exec randomization to point out when randomization takes place in both the per-boot and per-exec processes. Similarly, Figure 2 shows that *mmap()* system call allocates all the objects side by side in the *mmap_area* area during the per-object randomization taking place. The *rand()* provides random bits of desired length to the objects as shown in Figure 2.

Moreover, the degree of security provided by ASLR technique depends on the predictability of the random memory layout of a program and therefore, ‘non-reproducibility’ of the random sequences used in ASLR needs to be additionally ensured. This particular requirement is also attended in the present work.

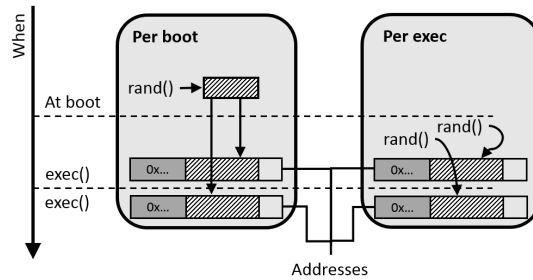


Figure 1: Per-boot versus Per-exec randomization [2]

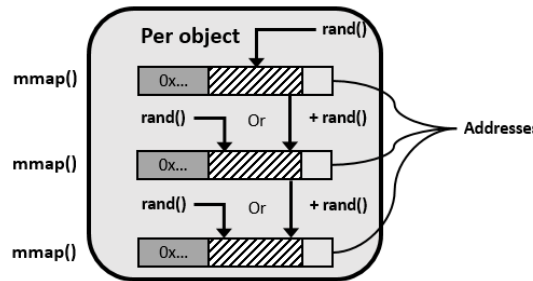


Figure 2: Per-object randomization [2]

Another important kernel application is the Morris-Thompson scheme [3, 8] which associates a n -bit random number with each password and concatenate and then encrypt together before storing it in the password file. A CSPRNG is used whenever a password is changed and a random number is required.

2.2. RNG requirements

Koc [1] and Schneier [9] collated the properties that various classes of RNG exhibit and formulated the following requirements:

1. $R1$: A random sequence generated by a RNG should have good statistical properties.

This requirement enables a RNG with a large period.

2. $R2$: A random sequence generated by a RNG should be unpredictable.

This requirement makes the prediction of the next bit infeasible in the stream, given the complete knowledge of the algorithm or hardware which generates the sequence and all of the previous bits in the stream. This gives the notion of Backward Secrecy.

3. $R3$: A random sequence generated by a RNG should not allow to compute previous internal state or values of the generator even if the internal state is known.

This gives the notion of Forward Secrecy.

4. $R4$: A random sequence generated by a RNG should not be reliably reproduced.

If the RNG is run twice with exactly the same input, it should produce two completely unrelated random sequences.

From definition, a PRNG meets only $R1$ requirement whereas CSPRNG meets $R1$, $R2$ and $R3$ requirements of RNG [9]. However, a TRNG meets $R2$, $R3$ and $R4$ requirements of the RNG [9]. In this paper, the proposed KCS-PRNG is designed in such a way that it meets the $R1$, $R2$ and $R3$ requirements along with the $R4$ requirement of RNG to a practical extent.

3. Cryptographically secure random number generators for kernels

Linux and Android kernels use `/dev/random` and `/dev/urandom` which are considered as CSPRNG i.e. the PRNG with inputs (meeting the requirement $R2$) for randomness generation. The limitations of these CSPRNGs are that they do not have enough entropy in the pool and they are not generating keys larger than the hash function that they used internally [10]. `/dev/random` keeps awaiting for the entropy pool to get sufficiently filled in, which results in diminished performance of the generator. `/dev/random` meets the RNG requirements $R1$, $R2$ and $R3$ but does not meet the $R4$ requirement. Though `/dev/urandom` has provision for unblocked fast supply of random sequences through unblocking pool of entropy but faces predictability issues [11]. `/dev/urandom` meets the requirements $R1$ and $R3$ but does not meet the requirement $R2$ and $R4$.

Yarrow [12] is a PRNG with true random inputs used by MacOS/iOS/FreeBSD kernels. This CSPRNG is too complex and under-specified in entropy handling context and also slow to provide an initial seed [10]. It uses Triple Data Encryption Standard (DES) block cipher for pseudorandom bitstream generation. Like `/dev/random`, Yarrow meets the requirements $R1$, $R2$ and $R3$ but does not meet the requirement $R4$.

Fortuna [13] is a popular CSPRNG and a refinement over Yarrow, used by the Windows kernel which uses its entropy effectively. It uses Advanced Encryption Standard (AES)-like cipher for the generator with 256-bit size of the block cipher key and a 128-bit counter. Fortuna produces a very good throughput of 20 clock cycles per byte on CPU type PC [13] and 7.2 Mbps throughput in software [14]. Fortuna implicitly accumulates entropy through hash, partitions the incoming entropy into multiple entropy pools and uses its pools at different rate for output generation in order to guarantee that at least one pool will remain available for use [4]. Though Viega [10] observed that it completely foregoes the entropy estimation and Fortuna and Yarrow both do not exhibit information-theoretic security. Like Yarrow, Fortuna also meets the requirements $R1$, $R2$ and $R3$ but does not meet 'non-reproducibility' i.e., the requirement $R4$.

It is imperative to note that the present kernel CSPRNGs do not meet the requirement of 'non-reproducibility' i.e., the requirement $R4$ which is a crucial feature that helps to prevent the kernel better from exploitation as discussed in Section 2.1. In this work, the proposed KCS-PRNG is designed in such a way that all the four requirements ($R1$ to $R4$) of an ideal RNG are met to ensure better prevention of the kernel from exploitation.

4. The proposed design of KCS-PRNG

Generation of high quality cryptographically secure pseudorandom bitstreams is an intricate task which needs efficient design of the generator taking statistical properties of randomness ($R1$), unpredictability ($R2$, $R3$) and non-reproducibility ($R4$) of the output bitstreams into consideration. For this reason, the proposed KCS-PRNG binds two modules in its design: first, a combination of two cryptographically safe elliptic curves and a nonlinear Sequence Generator consisting of nine clock-controlled LFSRs in alternating step configuration. Following are the design decisions and assumptions of the proposed KCS-PRNG:

4.1. Selection of elliptic curves

The main motivation of using elliptic curves in the proposed KCS-PRNG instead of stream ciphers/block ciphers like ChaCha20 and Triple DES or AES respectively as used by `/dev/(u)random` [10], Yarrow [12] and Fortuna [13] respectively is that one can choose different points on the selected elliptic curve to generate completely unrelated bitstreams under identical start conditions. Hence, the combination of elliptic curves and clock-controlled LFSRs in the proposed KCS-PRNG generates non-reproducible cryptographically secure pseudorandom bitstreams. Moreover, the combination of elliptic curve and LFSR has been proven to exhibit enhanced randomness properties [15]. Two elliptic curves are used in KCS-PRNG for added complexity where each elliptic curve provides nearly 2^{128} key space. The advantages of keeping elliptic curves with the clock-controlled LFSRs are twofold: first, the elliptic curves are used for mitigating the gap of ‘non-reproducibility’ property ($R4$) by the proposed method of replacing them periodically from a look-up table. Second, elliptic curves are used to generate bitstreams which are non-invertible due to underlying hard Elliptic Curve Discrete Logarithm Problem (ECDLP) and hence, they make the proposed KCS-PRNG provably secure as well as forward secure to resist backtracking attacks. However, the choice of elliptic curves is considered to be a randomly generated one rather than the standard elliptic curves with fixed coefficients as being recommended by agencies like NIST [16], Brainpool [17] etc., so that a look-up table can be created consisting of reasonably large number of cryptographically secure elliptic curves of one’s choice. The random derivation of elliptic curve parameters ensures trust and transparency in the implementation of elliptic curves [18]. The details of the two elliptic curves selected for use in the KCS-PRNG are presented in Section 7. One can create a look-up table consisting of elliptic curves of 256 bit field order of one’s choice for use in the KCS-PRNG. The discussion on generation mechanism of elliptic curves is outside the scope of this paper due to space limitation.

4.2. Selection of clock-controlled LFSRs

The proposed KCS-PRNG is targeted for integration in the operating system kernel and therefore, it is implemented in software. However, implementation of LFSR in software is slower than its hardware implementation [9, 19]. To address this performance issue, the Galois scheme is selected for optimal performance gain by the LFSRs in software without compromising the LFSR period and its cryptographic properties [9]. The chosen Galois configuration also saves excess operations as all the XOR operations are performed as a single operation [9]. A nonlinear Sequence Generator consisting of

nine LFSRs $L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8$ and L_9 with corresponding primitive polynomial degrees 29, 31, 37, 41, 43, 47, 53, 59 and 61 respectively is designed. The primitive polynomials for these LFSRs feedback functions are

$$\begin{aligned}
 L_1 &= x^{29} + x^{25} + x^{21} + x^{17} + x^{14} + x^{10} + x^6 + x^3 + 1, \\
 L_2 &= x^{31} + x^{27} + x^{23} + x^{19} + x^{15} + x^{11} + x^7 + x^3 + 1, \\
 L_3 &= x^{37} + x^{32} + x^{27} + x^{23} + x^{18} + x^{13} + x^9 + x^5 + 1, \\
 L_4 &= x^{41} + x^{36} + x^{31} + x^{26} + x^{20} + x^{15} + x^{10} + x^5 + 1, \\
 L_5 &= x^{43} + x^{37} + x^{31} + x^{25} + x^{20} + x^{15} + x^{10} + x^5 + 1, \\
 L_6 &= x^{47} + x^{41} + x^{35} + x^{29} + x^{23} + x^{17} + x^{11} + x^5 + 1, \\
 L_7 &= x^{53} + x^{46} + x^{40} + x^{33} + x^{26} + x^{19} + x^{13} + x^7 + 1, \\
 L_8 &= x^{59} + x^{52} + x^{44} + x^{36} + x^{29} + x^{22} + x^{14} + x^7 + 1, \\
 L_9 &= x^{61} + x^{53} + x^{45} + x^{38} + x^{30} + x^{23} + x^{15} + x^7 + 1.
 \end{aligned}$$

These primitive polynomials used by the nine LFSRs have uniformly distributed feedback coefficients selected from [20]. These nine LFSRs L_1, L_2, \dots, L_9 are further divided into three groups called Sequence Generator 1 (SG_1), Sequence Generator 2 (SG_2) and Sequence Generator 3 (SG_3). SG_1 has three LFSRs L_1, L_2 and L_3 whose output streams x_1, x_2 and x_3 are combined nonlinearly using nonlinear function

$$y_1 : f(x_1, x_2, x_3) = x_1x_2 \oplus x_2x_3 \oplus x_3x_1 \tag{1}$$

The resulting sequence y_1 has period $(2^{L_1} - 1)(2^{L_2} - 1)(2^{L_3} - 1)$ and linear complexity $(L_1L_2 + L_2L_3 + L_1L_3)$. Similarly, the period and linear complexity of the sequence y_2 generated from SG_2 using L_4, L_5, L_6 are $(2^{L_4} - 1)(2^{L_5} - 1)(2^{L_6} - 1)$ and $(L_4L_5 + L_5L_6 + L_6L_4)$ respectively whereas the period and linear complexity of the sequence y_3 generated from SG_3 using L_7, L_8, L_9 are $(2^{L_7} - 1)(2^{L_8} - 1)(2^{L_9} - 1)$ and $(L_7L_8 + L_8L_9 + L_9L_7)$ respectively. It may be noted that the initial state bits of all LFSRs together are $\sum_{i=1}^9 L_i = 401$ bits.

SG_1, SG_2 and SG_3 are configured in alternating step scheme to provide high linear complexity and large period of the Sequence Generator [21]. SG_1 is considered as the Controller of the Sequence Generator in the alternating step mode. It is known that the linear complexity $LC(x)$ of the overall alternating step generator is bounded as follows [21]:

$$(LC_2 + LC_3)^{2LC_1-1} < LC(x) \leq (LC_2 + LC_3)^{2LC_1} \tag{2}$$

where LC_1, LC_2 and LC_3 are the linear complexities of SG_1, SG_2 and SG_3 respectively. The Alternating Step Sequence Generator used in the proposed KCS-PRNG is depicted in Figure 3 and described in Algorithm 1 [21].

4.3. The proposed novel KCS-PRNG architecture

The proposed KCS-PRNG architecture is shown in Figure 4. The KCS-PRNG uses a Field converter, Elliptic curve Point Multiplication and a Selector in addition to the Sequence Generator and two elliptic curves in its design.

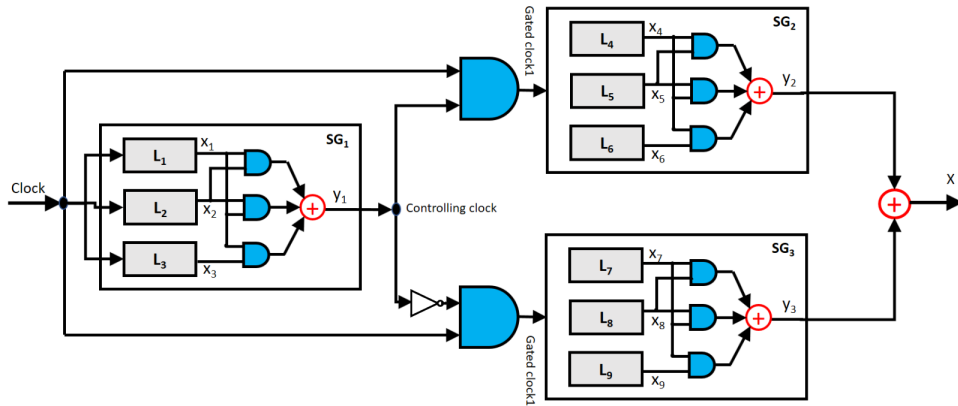


Figure 3: Sequence Generator in Alternating Step Configuration

Algorithm 1: Alternating Step Sequence Generator using Clock-controlled LFSRs**Input :** Sequence Generators SG_1, SG_2 and SG_3 **Output:** bit length n **for** $i \leftarrow 1$ **to** n **do** SG_1 is clocked **if** $SG_1 == 1$ **then** SG_2 is clocked. /* SG_3 is not clocked but its previous output bit is repeated. In case of the first clock cycle, previous output bit of SG_3 is taken as 0. */ **else** SG_3 is clocked /* SG_2 is not clocked but its previous output bit is repeated. In case of the first clock cycle, previous output bit of SG_2 is taken as 0. */ **end** **return** $y_2 \oplus y_3$ // Output of Sequence Generator in alternating step**end**

The two elliptic curves are selected using the procedure as shown in Algorithm 2. A look-up table \mathcal{T} with tuples (EC, EC_ID_Status) is created where EC is the elliptic curve and EC_ID_Status is the flag value to mark 0 for ‘un-used curve’ and 1 for the ‘used curve’. \mathcal{T} consists of 256 elliptic curves initially which are randomly generated and are cryptographically secure non-standard curves. All elliptic curves in \mathcal{T} are initially marked with $EC_ID_Status = 0$. On each reboot of the proposed KCS-PRNG, it picks up two elliptic curves from \mathcal{T} using Algorithm 2 and sets the corresponding

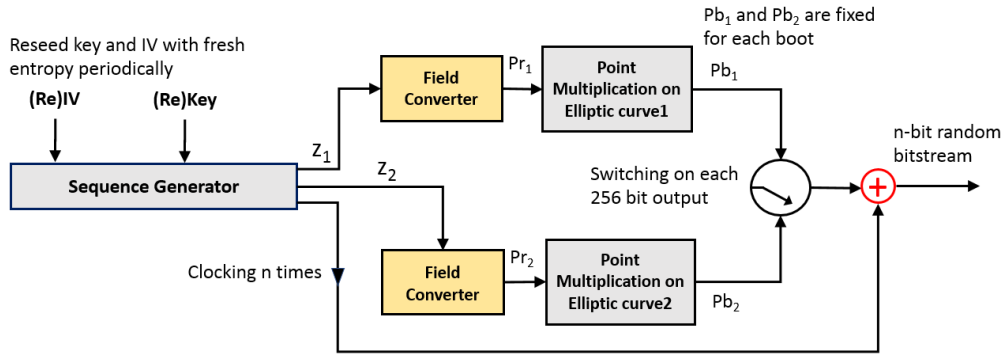


Figure 4: The proposed KCS-PRNG Architecture

Algorithm 2: Selection of 2 Elliptic curves

Input : Look-up Table $\mathcal{T}(EC_n, EC_n_ID_Status)$ where n is number of elliptic curves

Output: Elliptic curves EC_r, EC_s from \mathcal{T} where $r, s \in [1, n]$ and $r \neq s$

Count $n //$ Elliptic curves with $EC_n_ID_Status = 0 \forall n$ in \mathcal{T}

if $n \geq 2$ **then**

Fetch EC_r, EC_s from \mathcal{T} where $EC_r_ID_Status = 0$ and $EC_s_ID_Status = 0$

Set $EC_r_ID_Status \leftarrow 1, EC_s_ID_Status \leftarrow 1$

Update \mathcal{T}

return EC_r, EC_s

else

Set $EC_n_ID_Status = 0 \forall n$ in \mathcal{T}

Go to previous step

end

$EC_ID_Status = 1$ of both the used elliptic curves in \mathcal{T} . The advantage of \mathcal{T} is that even if the same seed (entropy) is supplied to the proposed KCS-PRNG on reboot of the generator, two new elliptic curves with $EC_ID_Status = 0$ will be selected from \mathcal{T} . The change of elliptic curves on each reboot of the KCS-PRNG changes the final output by altering the masking value between the output bits of the elliptic curves and the Sequence Generator. Hence, entirely unrelated bitstream are obtained as the output of the proposed generator even using exactly the same seed as input. When all elliptic curves in \mathcal{T} are used then EC_ID_Status flags are reset to 0 for all elliptic curves in \mathcal{T} in order to maintain unblocked supply of elliptic curves to the KCS-PRNG. More elliptic curves can be inserted into \mathcal{T} to consistently mitigate the requirement of ‘non-reproducibility’ property $R4$ of the KCS-PRNG. Here, the mitigating factor of the the RNG requirement $R4$ is directly proportional to the number of un-used elliptic curves available in \mathcal{T} . This idea makes the proposed KCS-PRNG to mitigate the RNG requirement $R4$ to a practical extent.

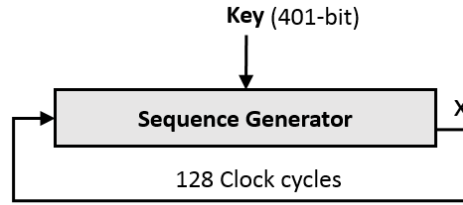


Figure 5: Initialization Stage 1: Loading and diffusion of the key

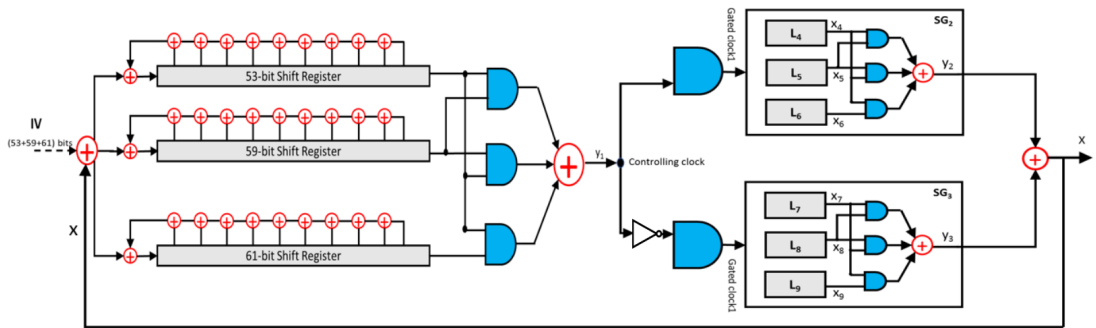


Figure 6: Initialization Stage 2: Loading of IV

4.4. Initialization of KCS-PRNG

The proposed KCS-PRNG uses two phases of pseudorandom bitstreams generation. In the first phase, the Sequence Generator is initialized whereas in the second phase, the desired number of bits of the pseudorandom sequence are generated using the Sequence Generator and the elliptic curves. The initialization phase involves two stages which includes, first, loading the key and initialization vector (IV) into the generator and second, diffusing the key-IV pair across the entire states of the Sequence Generator [22] as shown in Figure 5 and Figure 6 described in the Algorithm 3.

Algorithm 3 takes 574-bit of entropy bits which are harvested from various physical non-deterministic noise sources and generates 401-bit of key and 173-bit of Initialization Vector (IV). The key is first parallelly loaded into SG_1, SG_2 and SG_3 of the Sequence Generator. It is ensured that all the Most Significant Bits (MSBs) of L_1, L_2 and L_3 will be set to 1. The Sequence Generator is then clocked 128 times so that the key is diffused across the entire states of all the nine LFSRs L_1, L_2, \dots, L_9 and a new state of the Sequence Generator is obtained as shown in Figure 5. Further, a 173-bit IV is loaded into L_1, L_2 and L_3 of SG_1 in bitwise fashion by XORing with the feedback bit of the LFSR and the output bit of the Sequence Generator as shown in Figure 6. The Sequence Generator is once again clocked 128 times to diffuse the IV completely among the LFSRs in SG_1 and gets entirely new states of all the nine LFSRs. It is ensured that the MSBs of all the nine LFSRs L_1, L_2, \dots, L_9 are set to 1. Finally, the initialized Sequence Generator is returned.

Algorithm 3: Initialization of Sequence Generator

```

Input : 401-bit entropy for Key and 173-bit entropy for Initialization Vector (IV)
Output: Initialized Sequence Generator
// Stage 1: Loading LFSRs from the input Key
Initialize  $SG_1, SG_2$  and  $SG_3$  with 401-bit Key
if MSB of any LFSR == 0 then
  | Ensure MSB of LFSR as 1
end
// Stage 2: Diffusion of key into all LFSRs states
Clock Sequence Generator 128 times
// Stage 1: Loading 173-bit IV to  $SG_1$ 
for  $i \leftarrow 1$  to 173 do
  | Clock  $SG_1$  with feedback = Feedback bit  $\oplus$  IV bit  $\oplus$  output bit of Sequence Generator
end
// Stage 2: Diffusion of IV into all LFSRs states in  $SG_1$ 
Clock Sequence Generator 128 times
if MSB of the Sequence Generator == 0 then
  | Ensure MSB of the Sequence Generator as 1
end
return Initialized Sequence Generator

```

4.5. KCS-PRNG bitstream generation

The Sequence Generator generates two sequences z_1 and z_2 of 256-bit length each, which are used by the field converter as the inputs. The field converter transforms z_1 and z_2 into integers and then transforms them into the field elements P_{r_1} and P_{r_2} of the two elliptic curves. These field elements or the secrets P_{r_1} and P_{r_2} are given as inputs to the two elliptic curve point multiplication functions as described in Algorithm 4. The secrets P_{r_1} and P_{r_2} are multiplied with their corresponding base points G_1 and G_2 which yields a new point on each of the elliptic curves respectively. The x -coordinates of the two points obtained are the two integers P_{b_1} and P_{b_2} after transformation from the field elements. A selector is used to switch between the outputs of the two elliptic curves point multiplication functions to double the size of key space offered by the proposed KCS-PRNG.

Algorithm 4: Elliptic curve point multiplication

```

Input: Secrets  $P_{r_1}$  and  $P_{r_2}$  for 2 elliptic curves
Output: Points  $P_{b_1}$  and  $P_{b_2}$  of 2 elliptic curves in integer form
 $P_{b_1} \leftarrow G_1 \times P_{r_1}$  /*  $G_1$  is the base point selected on first elliptic curve and
   $P_{b_1}$  is the  $x$ -coordinate of the resultant point */
 $P_{b_1} \leftarrow \text{Integer}(P_{b_1})$  // Function to transform field to integer
 $P_{b_2} \leftarrow G_2 \times P_{r_2}$  /*  $G_2$  is the base point selected on second elliptic curve and
   $P_{b_2}$  is the  $x$ -coordinate of the resultant point */
 $P_{b_2} \leftarrow \text{Integer}(P_{b_2})$ 
return  $P_{b_1}, P_{b_2}$ 

```

Algorithm 5: The proposed KCS-PRNG bitstream generation

```

Input: Desired length of bitstream  $n$ ,  $(574 \times r)$ -bit entropy for key and IV where  $r = \lceil \frac{n}{100000} \rceil$  = number of
(re)seeding required for KCS-PRNG
Output:  $n$ -bit cryptographically secure pseudorandom bitstream
Run Algorithm 2 to select two elliptic curves from  $\mathcal{T}$ 
// Perform (Re)seeding to initialize the Sequence Generator
Run Algorithm 3 with input of 401-bit key and 173-bit IV to initialize the Sequence Generator
Run Algorithm 1 to generate 256-bit sequence  $z_1$ 
Transform  $z_1$  into field element  $P_{r_1}$  of first elliptic curve using field converter
Run Algorithm 4 with  $P_{r_1}$  as input to generate the integer  $P_{b_1}$ 
Run Algorithm 1 to generate 256-bit sequence  $z_2$ 
Transform  $z_2$  into field element  $P_{r_2}$  of second elliptic curve using field converter
Run Algorithm 4 with  $P_{r_2}$  as input to generate the integer  $P_{b_2}$ 
Set  $countSel = 1$ 
Set  $bitCount = 1$ 
Set  $t = 1$  where  $t = 1$  to  $\lceil \frac{n}{256} \rceil$ 
for  $i \leftarrow 1$  to  $\lceil \frac{n}{256} \rceil$  do
    if  $countSel == t \times 256$  then
        // Use Selector to select between the two elliptic curves
        if  $t$  is even then
            | Set  $el = P_{b_2}$ 
        else
            | Set  $el = P_{b_1}$ 
        end
         $countSel = 0$ 
         $t^{++}$ 
    end
    Clock Sequence Generator 256 times to generate 256-bit sequence  $s$ 
    if  $n < 256$  then
        | return  $X \oplus i^{th}$  position of  $el$  from LSB ( $i = 0$ ) to MSB ( $i = 255$ ) where  $X$  is 1-bit output from
        | Sequence Generator and  $i = 0$  to 255 // Output of KCS-PRNG
    else
        | return  $el \oplus s$  // Output of KCS-PRNG
    end
     $i^{++}$ 
    if  $i == 255$  then
        |  $i = 0$ 
    end
     $countSel^{++}$ 
     $bitCount^{++}$ 
    if  $bitCount == j \times 100000$  where  $j = 1$  to  $r$  then
        |  $n = n - (j \times 100000)$ 
        |  $j^{++}$ 
        | // Reseed the KCS-PRNG on every 100000 bits of output
        | Go to reseeding step in the beginning
    end
end

```

Algorithm 5 describes the cryptographically secure pseudorandom bitstream generation scheme of the proposed KCS-PRNG. Initially, two elliptic curves with hard ECDLP are selected from \mathcal{T} . The

Sequence Generator is then initialized with 401-bit key and 173-bit IV as discussed in Algorithm 3. The Sequence Generator is used to generate 256-bit sequence z_1 by clocking 256 times. Further, z_1 is converted into the field element of the first elliptic curve and considered as the secret P_{r_1} . The integer P_{b_1} is generated by using elliptic curve point multiplication function taking the secret P_{r_1} as input. Similarly, the integer P_{b_2} is also generated from the second elliptic curve point multiplication function. The Sequence Generator continuously generates n -bit length sequences as bounded by $\lceil \frac{n}{256} \rceil$ times loop. The proposed KCS-PRNG uses a selector iteratively select among P_{b_1} and P_{b_2} . The Sequence Generator is then clocked 256 times to generate 256-bit sequence s . The integers P_{b_1} or P_{b_2} is masked with s to produce 256-bit output by the KCS-PRNG. If $n < 256$, then 1-bit output of the Sequence Generator is masked with 1-bit of P_{b_1} or P_{b_2} (as decided by the selector) traversing from its Least Significant Bit (LSB) to MSB and result is returned. Once MSB of the P_{b_1} or P_{b_2} is used, the masking of the output of the Sequence Generator starts from the LSB of the P_{b_1} or P_{b_2} once again in rotating fashion. The KCS-PRNG is reseeded on every 100000 bit of output to maintain backward secrecy as shown in Algorithm 5.

4.6. Assumptions

Following assumptions are made in the proposed design of KCS-PRNG:

- KCS-PRNG always maintains 574-bit initial entropy.
- KCS-PRNG expects high per-bit entropy ≈ 1 for initialization. The generation details of entropy used in KCS-PRNG is outside the scope of this work.
- The Key and IV are parts of the seed and hence, they are immediately shredded after use and is non-recoverable.
- The (Re)keying and (Re)IVing are done using different TRNGs or entropy harvesters using various different physical noise sources.
- Elliptic curves used in KCS-PRNG are randomly generated, cryptographically safe and trustworthy.
- Look-up Table \mathcal{T} has authorized access only.

5. Security analysis of the proposed KCS-PRNG

5.1. Linear complexity analysis

Let linear complexities of the Sequence Generators SG_1, SG_2 and SG_3 be LC_1, LC_2 and LC_3 respectively and following equation (1), are given by

$$\begin{aligned} LC_1 &= L_1L_2 + L_2L_3 + L_1L_3 = 3119 \\ LC_2 &= L_4L_5 + L_5L_6 + L_4L_6 = 5711 \\ LC_3 &= L_7L_8 + L_8L_9 + L_7L_9 = 9959 \end{aligned} \tag{3}$$

where L_1, L_2, \dots, L_9 are the lengths of the LFSRs.

Moreover, while SG_1 is clocked regularly, SG_2 and SG_3 are connected in alternating step configuration. Thus, following equation (2), the overall linear complexity (LC) of the scheme is given by

$$\begin{aligned} (5711 + 9959)^{2 \times 3119 - 1} < LC(x) \leq (5711 + 9959)^{2 \times 3119} \\ \implies 15670^{6237} < LC(x) \leq 15670^{6238} \end{aligned} \quad (4)$$

It is imperative to note that the Sequence Generator of the proposed KCS-PRNG exhibits exponentially large linear complexity as demonstrated in equation (4) and therefore, the proposed generator is resistant to the Berlekamp-Massey attack [21].

5.2. Correlations test

We conducted two correlation tests of random bitstreams generated by the proposed KCS-PRNG to verify non-correlation in the bitstream. The first test conducted was Serial or Autocorrelation test (*sstring - AutoCor* test) which measures the correlation between the bits with the lag d [23]. In this test, a n -bit string is generated by the KCS-PRNG at the first level and the test statistic is computed such that it has the binomial distribution with the parameters being approximately standard normal for

Table 1: Correlation test of the proposed KCS-PRNG.

sstring-AutoCor test	$N = 1, n = 1048513, r = 0, s = 32, d = 1$
Normal statistic	0.41
p-value of test	0.34
Number of bits used	1048544
Result	Passed the test
sstring-AutoCor test	$N = 1, n = 1048514, r = 0, s = 32, d = 2$
Normal statistic	0.80
p-value of test	0.21
Number of bits used	1048544
Result	Passed the test
sstring-HammingCorr test	$N = 1, n = 32768, r = 0, s = 32, L = 32$
Normal statistic	-0.56
p-value of test	0.71
Number of bits used	1048576
Result	Passed the test
sstring-HammingCorr test	$N = 1, n = 16384, r = 0, s = 32, L = 64$
Normal statistic	0.45
p-value of test	0.33
Number of bits used	1048576
Result	Passed the test
sstring-HammingCorr test	$N = 1, n = 8192, r = 0, s = 32, L = 128$
Normal statistic	1.57
p-value of test	0.06
Number of bits used	1048576
Result	Passed the test

large $n - d$. The restriction imposed were $r + s \leq 32$ and $1 \leq d \leq \lfloor \frac{n}{2} \rfloor$ where r be the number of MSBs which are eliminated from the output before applying the test, s be the MSBs chosen from each generated random number and N be second-level number of replications [23, 24]. The second test conducted was the Hamming Correlation test (*sstring - HammingCorr*) [25] was used to measure bitwise correlation in the random bitstream file of 1GB size generated by the proposed KCS-PRNG which was estimated to be 0.000034. The obtained correlation is very close to the ideal correlation value of 0.0 and thus, concludes that the proposed design of the KCS-PRNG has no correlation issues and their results are shown in Table 1.

5.3. Period analysis (validation of requirement R1)

The Sequence Generator used in the KCS-PRNG comprises of nine LFSRs whose lengths L_1, L_2, \dots, L_9 are coprime to each other. Hence, the period (P) of the Sequence Generator is given by

$$P = \prod_{i=1}^9 (2^{L_i} - 1) \quad (5)$$

As the output of the Sequence Generator is masked with the integer obtained from the x -coordinate of the public key of one of the two elliptic curves, therefore, the period \mathcal{P} of the proposed KCS-PRNG is given by

$$\mathcal{P} = \begin{cases} N_1 \times \left(\prod_{i=1}^9 (2^{L_i} - 1) \right) & \text{if } (n \leq 256) \\ (N_1 + N_2) \times \left(\prod_{i=1}^9 (2^{L_i} - 1) \right) & \text{if } (n > 256) \end{cases} \quad (6)$$

where n be the number of output bits and N_1, N_2 are the order of the two elliptic curves and let $N_1 < N_2$.

It is prudent from equation (6) that the period \mathcal{P} of the proposed KCS-PRNG approximately lies in the range $[N_1 \times 2^{401}, (N_1 + N_2) \times 2^{401}]$ per boot which enables the proposed KCS-PRNG to generate very large bitstream without compromising the statistical properties of randomness.

5.4. Key space analysis

It is evident from equation (5) that the Sequence Generator in KCS-PRNG has a period of 2^{401} and thus, provides 2^{401} key space in case the generator gets seeded once and no reseeding happens. Moreover, the KCS-PRNG also uses two elliptic curves which provides 2^{128} and 2^{256} key space for $n \leq 256$ and $n > 256$ bits of output respectively to impose a successful Pollard's rho attack to solve the ECDLP. Hence the key space offered by the proposed KCS-PRNG is given by

$$\mathcal{K} = \begin{cases} (2^{401} \times 2^{128})^r = 2^{529r} & \text{if } (n \leq 256) \\ (2^{401} \times 2^{256})^r = 2^{657r} & \text{if } (n > 256) \end{cases} \quad (7)$$

where r be the number of (re)seeding the KCS-PRNG and n be the number of output bits of the proposed KCS-PRNG.

It is imperative to note that the key space offered by the proposed KCS-PRNG depends on the number of times the KCS-PRNG (re)seeds itself in single boot and therefore, exhibits virtually infinite key space in the range $\mathcal{K} \in [2^{529}, \infty)$ which is quite higher than the safe key space threshold of 2^{128} as recommended by [6, 26]. Therefore, the proposed KCS-PRNG comfortably resists brute force attacks.

5.5. Cross layer attack on kernel PRNG

A practical attack [27] using the weakness in the Linux Kernel PRNG is discovered that allowed the hackers to mount the cross-layer attacks against the Linux kernel to retrieve the internal states of the PRNG. The internal states of the kernel PRNG were compromised due to the linearity, same set of instances being used by the applications of the kernel PRNG and partially re-seeding issues respectively. The attackers were able to extract data from one PRNG consumer (network protocols like IPv4/IPv6, UDP etc.) in one Open Systems Interconnection (OSI) layer and used them to exploit another PRNG consumer in difference OSI layer. This weakness in the PRNG also allowed hackers to identify and track both the Linux and the Android devices. The compromised kernel was then used to downgrade E-mail security, hijack E-mails, hijack Hyper Text Transport Protocol (HTTP) traffic, circumvent E-mail anti-spam and blacklisting mechanisms, mount a local Denial of Service (DoS) attack (blackhole hosts), poison reverse Domain Name Server (DNS) resolutions and attack the machine's Network Time Protocol (NTP) client which is responsible for the machine's clock.

It is imperative to note that the compromised internal states of the kernel PRNG enabled the attackers to predict entire random sequences generated by it. However, the proposed KCS-PRNG does not allow such leakage of its internal states due to its unique design that leverages very high degree of non-linearity (as given in equation (4)) of the generator and generates non-reproducible random bit sequences to provide entirely unrelated pseudorandom sequences for each user applications.

6. Experimental validation of the proposed KCS-PRNG

6.1. Experimental validation of requirement $R1$

i. NIST statistical test results

NIST test suite consists of 15 statistical tests to certify statistical strength of randomness of the RNG. An output bitstream of 1GB file size is generated by the proposed KCS-PRNG and subjected to the NIST tests using NIST statistical test suite SP 800-22 version 2.1.2 [28]. The input block size was set to be 1000000 bits and 1000 bitstreams. The significance level α was selected as 99% to conduct the test. The proposed KCS-PRNG passed all the NIST statistical tests and the details of test results obtained are depicted in Table 2.

The p-value measures randomness and supposed to be greater than 0.01 i.e., the confidence level to conclude that the sequence is uniformly distributed whereas the proportion i.e., the minimum pass rate for the test should fall in the range [0.98056, 0.99943] having the confidence

interval $\alpha=0.01$ and 1000 bitstreams [7]. As indicated in Table 2, the proposed KCS-PRNG not only qualifies the pass rate threshold of 0.98056 but also reports better pass rate of 0.9896 as compared to the pass rates of 0.987 and 0.9887 reported by the TRNG [7] and the PRNG [6] respectively.

Table 2: NIST test results of the proposed KCS-PRNG output bitstreams of 1GB file size with the input of 1000000-bit block size and 1000 bitstreams.

Statistical Test	<i>P – value</i>	Proportion	Result
Frequency	0.737915	0.991	Pass
Block Frequency	0.591409	0.988	Pass
CumulativeSums*	0.680755	0.993	Pass
Runs	0.281232	0.992	Pass
Longest Run	0.526105	0.996	Pass
Rank	0.036113	0.996	Pass
FFT	0.103138	0.990	Pass
NonOverlappingTemplate*	0.794391	0.990	Pass
Overlapping	0.779188	0.987	Pass
Universal	0.773405	0.991	Pass
Approx Entropy	0.653773	0.989	Pass
RandomExcursions*	0.489508	0.983	Pass
RandomExcursionsVariant*	0.163362	0.985	Pass
Serial*	0.680755	0.988	Pass
Linear Complexity	0.682823	0.985	Pass

*Only the result of first test instance is indicated here from the original results due to limitation of space

ii. Diehard test results [29]

Diehard version 3.31.1 tests conduct a series of statistical tests and determine the p-values of the output bitstreams. The p-values indicate deviation of bit prediction from ideally expected probability of half. The expected p-value of a test should be in the range [0.025, 0.975] [30]. The proposed KCS-PRNG passed all the diehard tests as shown in Table 3.

iii. TestU01 test results [23]

TestU01 is believed to impose the toughest tests to evaluate the statistical quality of random bitstreams [6]. The binary bitstream of 1GB file size generated by the proposed KCS-PRNG is subjected to the Rabbit and Alphabit test batteries of TestU01. The Rabbit and the Alphabit, by default, selected 1048576 bits (2^{20} bits) for SmallCrush (a fast statistical test battery) evaluation and applied 38 and 17 statistical tests respectively to the proposed KCS-PRNG output bitstream. The output bitstreams of KCS-PRNG are found to have p-values within the acceptable range of [0.001, 0.999] [30] which proved that the proposed KCS-PRNG exhibits long period, good structure and non-linearity.

Table 3: Diehard test results of the proposed KCS-PRNG output bitstreams of 1GB file size.

test-name	ntup	tsamples	psamples	p-value	Assessment
diehard-birthdays	0	100	100	0.27561288	Passed
diehard-operm5	0	1000000	100	0.13184067	Passed
diehard-rank-32x32	0	40000	100	0.44295780	Passed
diehard-rank-6x8	0	100000	100	0.88076181	Passed
diehard-bitstream	0	2097152	100	0.42947798	Passed
diehard-opso	0	2097152	100	0.12604767	Passed
diehard-oqso	0	2097152	100	0.94641900	Passed
diehard-dna	0	2097152	100	0.24390543	Passed
diehard-count-1s-str	0	256000	100	0.62287409	Passed
diehard-count-1s-byt	0	256000	100	0.91047395	Passed
diehard-parking-lot	0	12000	100	0.79390338	Passed
diehard-2dsphere	2	8000	100	0.17731451	Passed
diehard-3dsphere	3	4000	100	0.45129204	Passed
diehard-squeeze	0	100000	100	0.53561994	Passed
diehard-sums	0	100	100	0.94209561	Passed
diehard-runs*	0	100000	100	0.14811353	Passed
diehard-craps*	0	200000	100	0.92115680	Passed
marsaglia-tsang-gcd*	0	10000000	100	0.53120802	Passed
sts-monobit	1	100000	100	0.64501072	Passed
sts-runs	2	100000	100	0.94961272	Passed
sts-serial*	1	100000	100	0.62077367	Passed
rgb-bitdist*	1	100000	100	0.95378266	Passed
rgb-minimum-distance*	2	10000	1000	0.87517368	Passed
rgb-permutations*	2	100000	100	0.75286377	Passed
rgb-lagged-sum*	0	1000000	100	0.00308570	Passed
rgb-kstest-test	0	10000	1000	0.03414230	Passed
dab-bytedistrib	0	51200000	1	0.17158919	Passed
dab-dct	256	50000	1	0.07312246	Passed
dab-filltree*	32	15000000	1	0.61801753	Passed
dab-filltree2*	0	5000000	1	0.69361846	Passed
dab-monobit2	12	65000000	1	0.42742922	Passed

*Only the result of first test instance is indicated here from the original results due to limitation of space

6.2. Validation of requirements R2 and R3

i. Next bit test

This test states that if a sequence of m -bits is generated by a generator, there should not be any feasible method which can predict the $(m + 1)$ th bit with the probability significantly higher than half [31, 32]. This test is associated with predictability of the successive bits generated by the KCS-PRNG.

Since the KCS-PRNG is reseeded with fresh additional entropy of 574 bits (401 bits of key and 173 bits of IV), therefore, it maintains backward security [13].

ii. Test for state compromise extension attacks

This test states that if some state of a generator is leaked at a given time to an attacker, it would not be possible to recover unknown PRNG outputs from that known state [33]. Fundamentally, the state compromise extension imposes two kinds of attack: first, a backtracking attack to learn previous outputs of the generator knowing some internal state of the generator at a particular time and second, the permanent compromise attack which enables all the future and past states of the generator vulnerable with the knowledge of some state at a given time [33].

Since the proposed KCS-PRNG is forward secure and provably secure due to underlying ECDLP intractability, therefore, it is resistant to the backtracking attack. Furthermore, as discussed in the next bit test, the proposed KCS-PRNG is (re)seeded on every 100000 bits of output generation, therefore, it exhibits backward secrecy and thus, resists the permanent compromise attack as well.

- iii. Entropy Estimation (Experimental Validation of Requirement R2, R3) Entropy is the measurement of unpredictability or uncertainty. For an ideal TRNG, the expected entropy is 1 per bit which means that each bit i.e., '0' or '1' have equal proportion 0.5 in the file containing random bitstream [7]. The proposed KCS-PRNG is subjected to ENT tool [25] for estimation of the entropy of the KCS-PRNG generated 1GB file of random bitstream. The observed value of the entropy of output bitstream generated by the proposed KCS-PRNG is found to be 0.99999975 per bit which asserts that the design of KCS-PRNG maintains nearly an ideal unpredictability.

6.3. Experimental validation of requirement R4

6.3.1. Non-reproducibility test

The non-reproducibility test is conducted to validate if the RNG requirement *R4* is met by the proposed KCS-PRNG. This test is conducted by running the generator twice with exactly the same input and verifying if the output sequences are completely unrelated. Authors [7] have referred the non-reproducibility test as the restart test and they validated the first 20 bit output sequences of the generator six times under identical start conditions. Table 4 shows that the proposed KCS-PRNG has passed the non-reproducibility test six times by producing six completely unrelated 32 bits using the same inputs to the proposed generator.

Moreover, the KCS-PRNG uses two different elliptic curves on each boot and therefore, the output bitstream would be entirely unrelated even generated under identical start conditions. Hence, it is inferred that the proposed KCS-PRNG generates non-reproducible pseudorandom bitstreams, provided it maintains minimum number of un-used elliptic curves (i.e., $t + 1$ where $t \geq 1$ is the number of (re)boots made by the KCS-PRNG such that the generator gets at least two un-used elliptic curve on each (re)boot) in the look-up table consisting of elliptic curves.

Table 4: Non-reproducibility test of the proposed KCS-PRNG under identical start conditions.

Key Input (401-bit entropy)	1905119BCDC809077DB45D 1B3921DB5C06D11 C56C7FE B4F8EE935A2FB16B055281816 DFC551AC73C3BBF76EE26B13 0B8F5E68
IV Input (173-bit entropy)	190B6B491CDD9E97E6AB 26552990F5481183DEF9AE55
Check	First run of KCS-PRNG
32-bit Output	0101010011101111110001110100100
Check	Second run of KCS-PRNG
32-bit Output	0001001000010000111100111111110
Check	Third run of KCS-PRNG
32-bit Output	1100010111000110101110010111101
Check	Fourth run of KCS-PRNG
32-bit Output	0110101001011010101000010110101
Check	Fifth run of KCS-PRNG
32-bit Output	10110001000111011001101100011011
Check	Sixth run of KCS-PRNG
32-bit Output	01001100110010111100010011100110

Table 5: Details of first elliptic curve with verification details [34] used in the proposed KCS-PRNG

Elliptic curve parameter/Validation	Value
Equation Model	Short Weierstrass
Prime field p	0xEEAA0DB0A46CE48AFCD288C714939E4063E1D801C55D1118202C76798B62B483
Coefficient a	0x33866AAA5914BC27D9ED986D7AF431BD8FC217D8E07D5BA5E44C1A4A355C7DD4
Coefficient b	0xCAAA0537DF123F85EC185A991B7200396B996C7921E6A7E07F08ED2A4801B0CA2
Co-factor h	0x1
Base Point G_x	0x3FBE1FF3CC8A893B2B018CC7D3D61961233F87F66FCB257D21805D1327426DE9
Base Point G_y	0xC5B219E84B008A4CB36CDF05B44E95354913756FCD92251F90BFB0A4F4D84AD8
Rho	127.8 // Key space of $2^{127.8}$ for Pollard's Rho attack on ECDLP
$Twist - rho$	127.8
$Joint - rho$	127.8
$verify - isElliptic$	True // Ensuring elliptic curve
$verify - P_{r_1} isOnCurve$	True // Ensuring private key as an elliptic curve group element
$verify - P_{b_1} isOnCurve$	True // Ensuring public key as an elliptic curve group element
$verify - safeField$	True // Elliptic curve defined over a suitable prime field
$verify - safeEquation$	True // Ensuring short Weierstrass equation
$verify - safeBase$	True // Ensuring base point with prime order
$verify - safeRho$	True // Ensuring ECDLP security
$verify - safeTransfer$	True // Ensuring ECDLP security from MOV attacks
$verify - safeDiscriminant$	True // Ensuring cubic curve
$verify - safeRigid$	True // Ensuring elliptic curve is generated using explained procedure
$verify - safeTwist$	True // Ensuring twist of the elliptic curve is safe
$verify - safeCurve$	True // if and only if all the other validations return 'True'

Table 6: Details of second elliptic curve with verification details [34] used in the proposed KCS-PRNG

Elliptic curve parameter/ Validation	Value
Equation Model	Short Weierstrass
Prime field p	0xF2A284E729748EA8BE82173F13412FC257C42095408D706528F5D8964BF2E237
Coefficient a	0xB29C202E105FE4C7EE5DECAF48258BFAB2E890AF5D96DE4553D82C3EC5D03C06
Coefficient b	0xC36BBDD9EE50EF046EA1D4DA85300673531B323B013043F9DC97B2FDD6A807B4
Co-factor h	0x1
Base Point G_x	0x1216C78C1FB8707C6B7B2496226B6F13CE25347DD9283A36FA354D09E2CDF4C3
Base Point G_y	0xA0AC0431A50C5DA5D25DCA1026946A2AADA19756ED326DA85A203B4A0B2BE342
Rho	127.8 // Key space of $2^{127.8}$ for Pollard's Rho attack on ECDLP
$Twist - rho$	127.8
$Joint - rho$	127.8
$verify - isElliptic$	True // Ensuring elliptic curve
$verify - P_{r_1} isOnCurve$	True // Ensuring private key as an elliptic curve group element
$verify - P_{b_1} isOnCurve$	True // Ensuring public key as an elliptic curve group element
$verify - safeField$	True // Elliptic curve defined over a suitable prime field
$verify - safeEquation$	True // Ensuring short Weierstrass equation
$verify - safeBase$	True // Ensuring base point with prime order
$verify - safeRho$	True // Ensuring ECDLP security
$verify - safeTransfer$	True // Ensuring ECDLP security from MOV attacks
$verify - safeDiscriminant$	True // Ensuring cubic curve
$verify - safeRigid$	True // Ensuring elliptic curve is generated using explained procedure
$verify - safeTwist$	True // Ensuring twist of the elliptic curve is safe
$verify - safeCurve$	True // if and only if all the other validations return 'True'

7. Details of two elliptic curves used in the proposed KCS-PRNG

Elliptic curves over 256-bit prime fields whose ECDLPs are found to be hard and method of computation is transparent and trustworthy, are selected for use in the proposed KCS-PRNG. The elliptic curves are generated randomly over the 256-bit prime field size in order to build the trust as indicated in [18, 35, 36, 37]. The generation mechanism of cryptographically safe elliptic curves is referred from [34, 38, 39, 40, 41, 42, 43] and followed with the procedure suggested in [44] to achieve trusted security. The proposed KCS-PRNG uses two elliptic curves which are generated randomly and verified for their cryptographic security as per the recommendations given in [34]. The verification details against the criteria as suggested in [34] of the two elliptic curves selected for experimentation purposes in this work are summarized in Table 5 and Table 6 respectively. A look-up table \mathcal{T} used in the proposed KCS-PRNG is created with 256 such elliptic curves initially as discussed in Section 4.3.

8. Results

The security analysis carried out in Section 5 has proved that the proposed KCS-PRNG exhibits: higher security property (from RNG requirements $R1$ to $R4$), provably secure, very high per bit

entropy rate, minimal bitwise correlation, highly nonlinear with linear complexity $LC(x)$ bounded as $15670^{6237} < LC(x) \leq 15670^{6238}$, very large period in the range $[N_1 \times 2^{401}, (N_1 + N_2) \times 2^{401}]$ per boot where $N_1 < N_2$ being the order of two elliptic curves used, huge key space in the range $[2^{529}, \infty)$ and impressive throughput of 2.5 Megabits per second as discussed in Section 9 to generate uninterrupted cryptographically secure bitstreams.

The proposed KCS-PRNG passed all the tests of NIST, Diehard and TestU01 test suites along with other tests to validate statistical qualities of randomness, cryptographic security and non-reproducibility as discussed in Section 6. The NIST test also proved that the proposed KCS-PRNG exhibits impressive and the highest proportion i.e., the pass rate of 0.9896 as compared to the existing PRNG [6] with 0.9887 and TRNG [7] with 0.987 proportion values. The KCS-PRNG demonstrated to exhibit nearly an ideal 0.9999975 per bit entropy and minimal serial correlation of 0.000034 in its generated bitstream.

9. Performance analysis of the proposed KCS-PRNG

The proposed KCS-PRNG was run on Intel® Core™ i7-7700 CPU @ 3.60GHz processor. The source code of the KCS-PRNG is developed in C++ and extensively used CryptoPP version 8.2.1 library. The KCS-PRNG software program was run on Ubuntu version 16.04.1 with kernel version 4.15.0-96-generic. The KCS-PRNG program was (re)seeded on every 100000 bits output in generation of 1GB file of pseudorandom bitstream. It gave an impressive throughput of 2.5 Mbps in software which asserts its high throughput-oriented design. The proposed KCS-PRNG for kernel applications offers a better security by meeting all the RNG requirements from $R1$ to $R4$ as compared to the existing PRNG [6] and kernel CSPRNGs like/dev/random [10, 11], Yarrow [12], and Fortuna [13].

10. Comparison of proposed KCS-PRNG with recent CSPRNGs for kernel applications

The proposed KCS-PRNG is designed to meet all the requirements of a RNG as discussed in Section 2.2. The features of the proposed KCS-PRNG are compared with the popular CSPRNGs used by the current operating system kernels and a recently well acknowledged TRNG [7] in Table 7. The reason behind the comparison of KCS-PRNG with TRNG is that, it meets the RNG requirement $R4$ which a TRNG only meets. Table 7 also consolidates interesting comparison results of KCS-PRNG with an existing TRNG based on Oscillator-Rings [7].

The KCS-PRNG is compared with popular kernel CSPRNGs namely /dev/(u)random used by Linux and Android kernels, Yarrow used by MacOS/iOS/FreeBSD kernel and Fortuna used by Windows kernel respectively on the basis of various criteria related to cryptographic security, randomness tests and throughput to conclude their suitability for strategic applications such as kernel applications.

Table 7: Comparison of the proposed KCS-PRNG with recent Kernel CSPRNGs and TRNG

Criterion	/dev(u)random	Yarrow	Fortuna	KCS-PRNG	TRNG
Hard problem used	ChaCha20 Stream cipher	3DES	AES128 in counter mode	ECDLP	Physical property of Oscillator-Rings
Hash function	SHA160, MD5 [45]	SHA160	SHA256	SHA256	Not applicable
RNG requirements met	$R1, R2, R3$	$R1, R2, R3$	$R1, R2, R3$	$R1, R2, R3,$ $R4$ (Mitigated)	$R1, R2, R3, R4$
Unblocked supply of random bits	No	No	Yes	Yes	Yes
Correlation Test	*	*	*	Passed (serial correlation of 0.000034)	Passed
Per bit entropy rate	*	*	*	0.99999975	0.9993
Linear complexity $LC(x)$	*	*	*	$15670^{6237} < LC(x) \leq 15670^{6238}$	Not applicable
Period	*	*	2^{128} in single call [14]	$[N_1 \times 2^{401}, (N_1 + N_2) \times 2^{401}]$	Infinite
Key space	*	*	*	$[2^{529}, \infty)$	Infinite
Throughput	8-12Kbps [45]	No results [45]	7.2 Mbps	2.5 Mbps	6 Mbps on Xilinx Spartan-3A FPGA
Statistical tests passed	Diehard [45]	Not available [45]	Diehard [14]	NIST, Diehard, TestU01	NIST
NIST proportion obtained	*	*	*	0.9896	0.987
Restart/Non-reproducibility Test	*	*	*	Passed	Passed

*No reference available

11. Conclusion

The operating system kernel demands a high quality CSPRNG for its randomness requirements. A novel CSPRNG called KCS-PRNG is presented in this paper which exhibits qualities of a CSPRNG as well as of a TRNG (i.e., it also includes non-reproducibility of the generated random bitstreams) for use in kernel and in various other cryptographic applications. The combination of clock-controlled LFSRs as a nonlinear sequence generator and two non-standard and trusted elliptic curves is proven to be an excellent choice of designing a CSPRNG. An extensive security analysis of the proposed KCS-PRNG was performed which proved that the proposed generator is resistant to important attacks like Berlekamp-Massey attacks, brute force attacks, next-bit tests, state compromise extension attacks and correlation attacks on the proposed generator. The proposed design of the KCS-PRNG allows periodic change of elliptic curves in the elliptic curve look-up table maintained by the generator to mitigate the gap of the security property *R4* i.e., ‘non-reproducibility’ requirement to a practical extent for the first time in the literature. The use of elliptic curves from its look-up table makes the proposed KCS-PRNG customizable than the current popular kernel CSPRNGs like `/dev/random`, Yarrow and Fortuna whose designs are based on block ciphers like Triple DES and AES respectively. Hence, it is inferred that the proposed KCS-PRNG qualifies as a competent CSPRNG for adoption in the kernel applications.

Acknowledgements

The authors thank Society for Electronic Transactions and Security (SETS), Chennai for providing the research opportunity to carry out this proposed work. The authors show their deepest gratitude to Dr. P. V. Ananda Mohan and Dr. Reshmi T. R. for their inputs and anonymous reviewers for their review and Mr. T. Santhosh Kumar for help in experimentation. Authors also thank to Mr. Ritesh Dhote, Mr. Aditya Saha, Ms. Sonal Priya Kamal and Ms. Diya V. A. for help in final formatting.

References

- [1] Koç ÇK. About cryptographic engineering. In: Cryptographic engineering, pp. 1–4. Springer, 2009. doi:10.1007/978-0-387-71817-0_1.
- [2] Marco-Gisbert H, Ripoll Ripoll I. Address space layout randomization next generation. *Applied Sciences*, 2019. **9**(14):2928. doi:10.3390/app9142928.
- [3] Tanenbaum AS, Woodhull AS. Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series). Prentice Hall, 2006. ISBN:0131429388.
- [4] Dodis Y, Shamir A, Stephens-Davidowitz N, Wichs D. How to eat your entropy and have it too: Optimal recovery strategies for compromised RNGs. *Algorithmica*, 2017. **79**(4):1196–1232. doi:10.1007/978-3-662-44381-1_3.
- [5] Dörre F, Klebanov V. Pseudo-random number generator verification: A case study. In: VSSTE. Springer, 2015 pp. 61–72. doi:10.1007/978-3-319-29613-5_4.

- [6] Alhadawi HS, Zolkipli MF, Ismail SM, Lambić D. Designing a pseudorandom bit generator based on LFSRs and a discrete chaotic map. *Cryptologia*, 2019. **43**(3):190–211. doi:10.1080/01611194.2018.1548390.
- [7] Anandakumar NN, Sanadhya SK, Hashmi MS. FPGA-based true random number generation using programmable delays in oscillator-rings. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2019. **67**(3):570–574. doi:10.1109/TCSII.2019.2919891.
- [8] Silberschatz A. Instructor’s Manual to Accompany: Operating System Concepts. 2015.
- [9] Schneier B. Applied cryptography: protocols, algorithms, and source code in C. John Wiley & Sons, Second edition, 2007.
- [10] Viega J. Practical random number generation in software. In: 19th Annual Computer Security Applications Conference, 2003. Proceedings. IEEE, 2003 pp. 129–140.
- [11] Dodis Y, Pointcheval D, Ruhault S, Vergniaud D, Wichs D. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013 pp. 647–658. doi:10.1145/2508859.2516653.
- [12] Kelsey J, Schneier B, Ferguson N. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In: International Workshop on Selected Areas in Cryptography. Springer, 1999 pp. 13–33. doi:10.1007/3-540-46513-8.2.
- [13] Ferguson N, Schneier B, Kohno T. Cryptography engineering: design principles and practical applications. John Wiley & Sons, 2011. ISBN-10:0470474246, 13:978-0470474242.
- [14] McEvoy R, Curran J, Cotter P, Murphy C. Fortuna: cryptographically secure pseudo-random number generation in software and hardware. In: 2006 IET Irish Signals and Systems Conference. IET, 2006 pp. 457–462. ISBN-0-86341-665-9.
- [15] Gong G, Lam CC. Linear recursive sequences over elliptic curves. In: Sequences and their applications, pp. 182–196. Springer, 2002. doi:10.1007/978-1-4471-0673-9_13.
- [16] Kerry CF, Gallagher PD. Digital signature standard (DSS). *FIPS PUB*, 2013. pp. 186–4.
- [17] Brainpool E. Brainpool Standard Curves and Curve Generation, 2005.
- [18] Bernstein DJ, Chou T, Chuengsatiansup C, Hülsing A, Lambooj E, Lange T, Niederhagen R, Vredendaal Cv. How to Manipulate Curve Standards: A White Paper for the Black Hat <http://bada55.cr.jp.to>. In: International Conference on Research in Security Standardisation. Springer, 2015 pp. 109–139. doi:10.1007/978-3-319-27152-1_6.
- [19] Mukhopadhyay S, Sarkar P. Application of LFSRs for parallel sequence generation in cryptologic algorithms. In: International Conference on Computational Science and Its Applications. Springer, 2006 pp. 436–445. doi:10.1007/11751595_47.

- [20] Rajski J, Tyszer J. Primitive polynomials over GF (2) of degree up to 660 with uniformly distributed coefficients. *Journal of Electronic testing*, 2003. **19**(6):645–657. doi:10.1023/A:1027422805851.
- [21] Menezes AJ, Van Oorschot PC, Vanstone SA. Handbook of applied cryptography. CRC press, 2018. ISBN-0-8493-8523-7.
- [22] Teo SG. Analysis of nonlinear sequences and streamciphers. Ph.D. thesis, Queensland University of Technology, 2013.
- [23] L'ecuyer P, Simard R. TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 2007. **33**(4):1–40. doi:10.1145/1268776.1268777.
- [24] L'Ecuyer P, Simard R. TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators—User's Guide, Compact Version. 2013. doi:10.1145/1268776.1268777.
- [25] Walker J. ENT: a pseudorandom number sequence test program. *Software and documentation available at/www.fourmilab.ch/random/S*, 2008.
- [26] II E, Sym D. ECRYPT II. 2010.
- [27] Klein A. Cross layer attacks and how to use them (for dns cache poisoning, device tracking and more). In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021 pp. 1179–1196. doi:10.1109/SP40001.2021.00054.
- [28] Bassham III LE, Rukhin AL, Soto J, Nechvatal JR, Smid ME, Barker EB, Leigh SD, Levenson M, Vangel M, Banks DL, et al. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications, 2010.
- [29] Marsaglia G. DIEHARD Test suite. 1998. **8**(01):2014. Online:<http://www.stat.fsu.edu/pub/diehard/>. Last visited,
- [30] Bhattacharjee K, Maity K, Das S. A search for good pseudo-random number generators: Survey and empirical studies. 2018. *arXiv preprint arXiv:1811.04035*.
- [31] Lavasani A, Eghlidis T. Practical next bit test for evaluating pseudorandom sequences. 2009. ID:37894176.
- [32] GG Rose AG, Xiao L. Cryptographically secure pseudo-random number generator, U.S. Patent, 2011.
- [33] Kelsey J, Schneier B, Wagner D, Hall C. Cryptanalytic attacks on pseudorandom number generators. In: International workshop on fast software encryption. Springer, 1998 pp. 168–188. doi:10.1007/3-540-69710-1_12.
- [34] Bernstein DJ, Lange T, et al. SafeCurves: choosing safe curves for elliptic curve cryptography. 2014. **9**. URL <https://safecurves.cr.jp.to.Citationsinthisdocument>,

- [35] Shumow D, Ferguson N. On the possibility of a back door in the NIST SP800-90 Dual Ec Prng. In: Proc. Crypto, volume 7. 2007 .
- [36] Hales TC. The NSA back door to NIST. *Notices of the AMS*, 2013. **61**(2):190–192. doi:10.1090/NOTI1078.
- [37] Bernstein DJ, Lange T. Security dangers of the NIST curves. In: Invited talk, International State of the Art Cryptography Workshop, Athens, Greece. 2013.
- [38] Konstantinou E, Kontogeorgis A, Stamatiou YC, Zaroliagis C. On the efficient generation of prime-order elliptic curves. *Journal of cryptology*, 2010. **23**(3):477–503. doi:10.1007/s00145-009-9037-2.
- [39] Menezes AJ, Okamoto T, Vanstone SA. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on information Theory*, 1993. **39**(5):1639–1646. doi:10.1109/18.259647.
- [40] Cheng Q. Hard problems of algebraic geometry codes. *IEEE Transactions on Information Theory*, 2008. **54**(1):402–406.
- [41] Bos JW, Costello C, Longa P, Naehrig M. Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, 2016. **6**(4):259–286. doi:10.1007/s13389-015-0097-y.
- [42] Smart NP. The discrete logarithm problem on elliptic curves of trace one. *Journal of cryptology*, 1999. **12**(3):193–196. doi:10.1007/s001459900052.
- [43] Koblitz N, Menezes A, Vanstone S. Guide to elliptic curve cryptography. 2004. doi:10.1007/b97644.
- [44] Abhishek K, Raj EGDP. Computation of Trusted Short Weierstrass Elliptic Curves for Cryptography. *Cybernetics and Information Technologies*, 2021. **21**(2):70–88. doi:10.2478/cait-2021-0020.
- [45] Röck A. Pseudorandom number generators for cryptographic applications. 2005. doi:10.1007/0-387-23483-7_330.