**EUROPEAN JOURNAL OF PURE AND APPLIED MATHEMATICS**
Vol. 15, No. 4, 2022, 1426-1443
ISSN 1307-5543 – ejpam.com
Published by New York Business Global

# The Self-Shrinking Conflation Generator: A Proposed Improvement to the Self-Shrinking Generator

Vikram Kanth[1], Thor Martinsen[2], Pantelimon Stănică[2,*]

[1] *Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA 93943, USA*
[2] *Department of Applied Mathematics, Naval Postgraduate School, Monterey, CA 93943, USA*

**Abstract.** The backbone of many cybersecurity applications and algorithms require random numbers. One of the most commonly used pseudo-random number generators is the Linear Feedback Shift Register (LFSR), which is fast, computationally inexpensive, and has excellent statistical properties. Unfortunately LFSRs have a number of weaknesses, some of which were addressed by decimation-based sequence generators such as the self-shrinking generator (SSG). Regrettably, the SSG was also found to be vulnerable to attack. In this paper, we propose an improvement to the SSG called the self-shrinking conflation generator (SSCG). Our approach is based on the observation that what is discarded during the self-shrinking process of the SSG, is from a cryptographic perspective, just as good as that which is kept. By combining the bits the SSG would normally discard with those it retains, using the exclusive OR (XOR) operation, we create a modified SSG bitstream with several improved characteristics. To highlight these improvements, we provide some mathematical security analysis associated with this approach, apply the National Institute of Standards and Technology (NIST) statistical test suite to several different bitstreams created using LFSRs driven by different degree primitive polynomials, and compare our results to that of the SSG.

**2020 Mathematics Subject Classifications**: 94A05, 94A55, 94A60

**Key Words and Phrases**: Linear Feedback Shift Register, Lightweight Stream Cipher, Self-Shrinking Generator

## 1. Introduction and Background

Cryptographic algorithms are an essential element of many cyber defenses [22]. Cryptographic algorithms are used for tasks such as authentication, data encryption, and digital signatures, and are the most commonly used privacy protection method in the IoT domain [22, 27]. The security of any of these cryptographic algorithms is generally dependent

*Corresponding author.
DOI: https://doi.org/10.29020/nybg.ejpam.v15i4.4504

*Email addresses:* vkkanth@nps.edu (V. Kanth)
tmartins@nps.edu (T. Martinsen), pstanica@nps.edu (P. Stănică)

 

upon the underlying pseudorandom number generator (PRNG) used, since random numbers (random bits) are required as input for things like keying material, auxiliary quantities for digital signatures, or generating authentication protocol challenges [12]. Thus, there is a pervasive need for cryptographically secure pseudorandom number generators (CSPRNGs). Furthermore, in today's distributed and mobile computing environment, devices need inexpensive and secure cryptographic solutions in order to overcome resource constraints like limited processing power and battery life [21]. In this paper, we present a novel, computationally cheap, and (we claim) secure method to generate pseudorandom bits. We simply XOR the discarded sequence as well as the self-shrinking generator (SSG) of an $m$-sequence (see Sections 1.2 1.3, for details). We will refer to our generator as a self-shrinking conflation generator (SSCG). We first investigate the period and the linear complexity of the discarded sequence, and use these to find an upper bound for the period and linear complexity of the SSCG. We consider guess-and-determine attacks, as well as exhaustive search and entropy attacks and argue that these do not apply to SSCG. We finally use National Institute of Standards and Technology (NIST) statistical tests to further validate the properties of the generator, for some sample primitive polynomials, and display an example in the last appendix of NIST tests on 100 bitstreams of 5 million bits.

### 1.1. Some definitions

We take $\mathbb{F}_2$ and $\mathbb{F}_{2^n}$ to be the respective two-element (binary) field and the field of dimension $n$, and $\mathbb{F}_2^n$ to be the vector space of dimension $n$ over the binary field. The absolute trace is the map $\mathrm{Tr}_1^n : \mathbb{F}_{2^n} \to \mathbb{F}_2$ is

$$\mathrm{Tr}_1^n(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{n-1}},$$

for all $x \in \mathbb{F}_{2^n}$. It is known [16] that the trace is an $\mathbb{F}_2$-linear map, invariant under the Frobenius automorphism (the squaring $x \to x^2$, in the binary case).

### 1.2. LFSRs and $m$-sequences background

There are many different approaches to generate random numbers efficiently. One of the most commonly used approaches to pseudorandom bit generation is the LFSR, which has both excellent baseline statistical properties [7] and is very fast. To give some perspective on the ubiquity of LFSRs, a look at the specifications for 3G, LTE, Wi-Fi, GPS, and Bluetooth reveals how common LFSRs actually are [23]. LFSRs are presented in great detail in [7] but a small summary is presented here. LFSRs, at the most basic level are a set of registers in which bits shift every clock cycle. Those bits are combined with each other based on the mathematical relation defined by a polynomial of the form $f(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1x + 1$. If the polynomial $f(x)$ is primitive, the length of the sequence produced is maximal. These sequences are of length $2^n - 1$, where $n$ is the highest degree of the polynomial, and are referred to as a maximal length sequence, or $m$-sequence for short [7]. For example, take the $m$-sequence generated by running an LFSR driven by the primitive polynomial $f(x) = x^4 + x + 1$. A recurrence relation for this

polynomial is $a_4 = a_1 + a_0$. The $m$-sequence generated by the LFSR with an initial seed of 1111 is 111100010011010. This example can be found in [8].

Another important concept for our discussion is linear complexity (LC). Linear complexity is defined as the length of the shortest LFSR that can generate a given sequence [7, 8, 20]. Alternatively, the linear complexity of a sequence is also the least order of a homogeneous linear recurrence satisfied by that sequence. This fact leads to the primary weakness of a random sequence generated by an LFSR. An efficient method, called the Berlekamp-Massey algorithm, is capable of calculating the linear complexity of a sequence [5, 9]. In fact, given $2\ell$ bits or more of a sequence (that can be generated by a length $\ell$ recurrence), the Berlekamp-Massey algorithm recovers the underlying polynomial used in the LFSR.

Recall that a periodic sequence $s = \{s_i\}_i$ of period $T$ satisfies $s_{i+T} = s_i$. The smallest such $T$ is called a *least period*. We shall refer to the subsequence $s' = \{s_0, s_1, \ldots, s_{T-1}\}$ as a cycle of $s$, and if $T$ is the least period, then $s'$ is a *minimal cycle*.

We define the linear complexity more precisely below. For a periodic sequence $s = (s_0, s_1, \ldots, s_{T-1})^\infty$ (of least period $T$) over $\mathbb{F}_2$, we let $S(x) = s_0 + s_1 x + \cdots + s_{T-1}x^{T-1}$ be the polynomial corresponding to the sequence $s$. It is known [7, 20] that the sequence $s$ can be represented as the power series

$$\sum_{i\geq 0} s_i x^i = \frac{S(x)}{1-x^T} = \frac{g(x)}{f(x)},$$

where $\gcd(g(x), f(x)) = 1, \deg(g(x)) < \deg(f(x))$ (deg represents the degree of the corresponding polynomial). The *linear complexity* of $s$, denoted by $LC(s)$, is then

$$LC(s) = \deg\left(\frac{1-x^T}{\gcd(S(x), 1-x^T)}\right) = \deg(f(x)).$$

As one might expect, the linear complexity is less than or equal to the (least) period, more precisely

$$LC(s) = T - \deg(\gcd(S(x), 1-x^T)). \tag{1.1}$$

## 1.3. The Self-Shrinking Generator (SSG)

A number of different approaches have been used to increase the security of LFSRs. One of these approaches is decimation or filtering. The idea behind this approach is to remove bits of the generated sequence in order to destroy its structure [4]. The shrinking generator and its follow-on, the self-shrinking generator, were two approaches that have been extensively studied. We concern ourselves here with the self-shrinking generator. This generator is a special case of the shrinking generator. It only requires one LFSR to both produce the initial sequence and drive its decimation process [17]. A diagram depicting the SSG is shown in Figure 1. The original self-shrinking generator can be described as follows [17]: An input sequence $(a_0, a_1, a_2, \ldots)$ can be treated as a sequence of bit pairs $((a_0, a_1), (a_2, a_3), \ldots)$. Given a bit pair $(a_{2i}, a_{2i+1})$, if the bit $a_{2i} = 1$, then the

bit $a_{2i+1}$ is output. If $a_{2i}$ is zero, the bit pair $(a_{2i}, a_{2i+1})$ is discarded. While the input sequence to the SSG need not be an $m$-sequence from an LFSR, in this work, we use an $m$-sequence to drive the self-shrinking process.

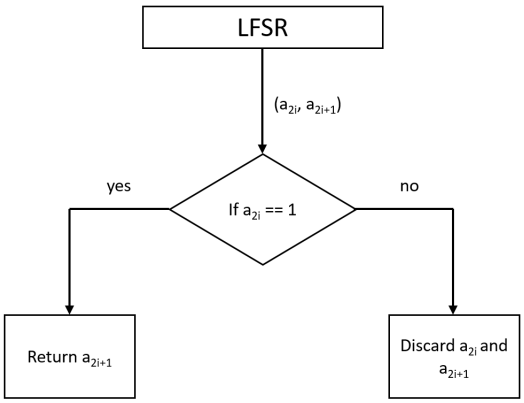


Figure 1: SSG implementation flowchart

For example, using the $m$-sequence repeated twice from our example LFSR in Section 1.2: 11110001001101011110001001101O, and applying the shrinking procedure described earlier, results in the output 11110000. It is important to note that while the algorithm specifies the bit pairs 10 and 11, it would be just as valid to use the bit pairs 00 and 01. In fact, one of the fundamental contributions of this paper is a comparison of using the SSG with the leading 0 bit pairs rather than the leading 1 bit pairs. In some cases, the SSG that uses the leading 0 bit pairs performs better. This will be detailed in Section 2. The SSG is a substantial improvement over a basic LFSR. Much research has been done into its properties and weaknesses. For the purpose of comparison, we present the bounds for the least period and the linear complexity of the SSG in Table 1.

| | Min | Max |
|---|---|---|
| Period | $2^{\lfloor n/2 \rfloor}$ | $2^{n-1}$ |
| LC | $2^{\lfloor n/2 \rfloor - 1}$ | $2^{n-1} - (n-2)$ |

Table 1: Bounds on the period and linear complexity of the self-shrinking generator from [1, 17]

There are a number of attacks that have been devised against the SSG, and we mention some of these next. These include exhaustive search and entropy attacks [17]. As time has passed, more sophisticated attacks have been developed to subvert the cryptographic properties of the SSG. These include a backtracking-based attack [24], a BDD (Binary Decision Diagram)-based attack [15], a long keystream attack [18], and the HJ-attack [10]. A formal comparison of our proposed approach against all of these classes of attacks is left for future work, though we present arguments for why some of these attacks are not possible in our construction.

### 1.4. Related prior work

There have been a handful of attempts to extend the SSG over the past several years. The first was the generalized self-shrinking generator (GSSG) [11]. Cryptanalysis of the GSSG showed that it was at most as secure as the SSG [26]. Follow-on attempts included the modified self-shrinking generator, which used groups of three bits to drive the discarding process [13] and its generalization, the t-modified shrinking generator [2]. Finally, Chang et al. proposed SSG-XOR, which used chunks of four bits, with the first two bits driving the discarding process [3]. A diagram of their process can be seen in Figure 2. Our idea uses a fundamentally different approach than those presented in this section.

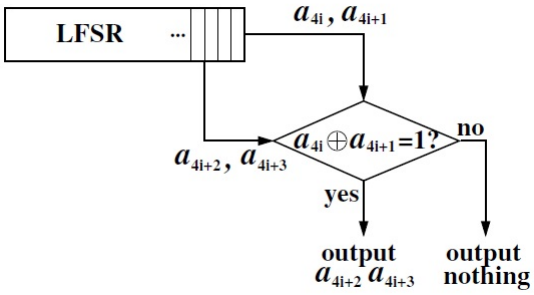


Figure 2: SSG-XOR diagram from [3]

## 2. The SSG using zeroes instead of ones

As was mentioned in Section 1.3, the original SSG algorithm uses the bit pairs 10 and 11. There is not much in the literature regarding using the bit pairs 00 and 01 to drive the SSG instead of the bit pairs 10 and 11. In fact, using the bit pairs 00 and 01 to drive the SSG sometimes results in sequences that can perform better than their 1 bit pair counterparts. Table 2 shows a comparison of both approaches using LFSRs driven by the various polynomials listed below. A more complete table with several primitive polynomials up to degree 15 can be found in Appendix 6. From this point onward, we shall be using the notations 1-SSG and 0-SSG to distinguish which first bit was used to drive the decimation process (1 and 0 respectively).
These examples reveal a few important facts. For a given polynomial, the linear complexity of either approach need not be the same. Additionally, either approach can have a larger linear complexity than the other. Finally, the bounds on the linear complexity of an SSG sequence given by [1] do not describe the bounds of the SSG sequence driven by the bit pairs 00 and 01. We provide an analysis of both the least period and linear complexity of this SSG approach in the following sections.

One might be tempted to believe that the 0-SSG corresponding to an $m$-sequence $s$ is simply the complement of the 1-SSG corresponding to $\bar{s}$. However, that is not true, as one can easily see by working out some examples. Let the minimal cycle of

| Polynomial | 1-SSG LC | 0-SSG LC |
|---|---|---|
| $x^5+x^4+x^2+x+1$ | 13 | 14 |
| $x^6+x^5+x^3+x^2+1$ | 28 | 30 |
| $x^7+x+1$ | 57 | 62 |
| $x^8+x^4+x^3+x^2+1$ | 121 | 126 |
| $x^9+x^4+1$ | 249 | 254 |
| $x^{10}+x^3+1$ | 504 | 507 |
| $x^{11}+x^5+x^3+x+1$ | 1013 | 1022 |
| $x^{12}+x^6+x^4+x+1$ | 2037 | 2035 |
| $x^{13}+x^4+x^3+x+1$ | 4083 | 4092 |

Table 2: Comparison of SSG approaches using either zeroes or ones

the $m$-sequence $s$ (as in Section 1.3) be $S_1 = 11110001001101011110001001101 0$. The minimal cycle of the 0-SSG is then $p_0 = 0101101$ and of the 1-SSG is $p_1 = 11110000$ (observe that we have maximum periods, for this example). Complementing, we get $\bar{S}_1 = 00001110110010101000011101100101$ with the minimal cycle of the corresponding 0-SSG being $p_0' = 00001111 = \bar{p}_1$, and the minimal cycle of the 1-SSG being $p_1' = 01000$, and so $p_0$ is not recoverable by the complementation method.

The $m$-sequence $z$ can be described as the absolute trace $s_i = \mathrm{Tr}_1^n(c\alpha^i)$, where $c \in \mathbb{F}_{2^n}$ (determined by the sequence $s$) and $\alpha$ is a primitive element of the field $\mathbb{F}_{2^n}$. The elements of the 1-SSG, respectively, 0-SSG are then

$$z_i = s_{2\tau(i)+1},$$

where $\tau(i)$ is the unique nonnegative integer such that $s_{2\tau(i)} = 1$ or $s_{2\tau(i)} = 0$ respectively, and there are $i+1$ elements that are one or zero respectively in the sequence $s_0 s_1 \dots s_{2\tau(i)}$. Expressing it using the absolute trace, we obtain $s_{2j} = \mathrm{Tr}_1^n(c\alpha^{2j}) = \mathrm{Tr}_1^n\left((c^{2^{n-1}}\alpha)^2\right) = T(\alpha^j)$, where $T$ is the nonzero $\mathbb{F}_2$-linear map, defined by $T(x) = \mathrm{Tr}_1^n\left(c^{2^{n-1}}x\right)$ (we used here the fact that the trace is invariant under the Frobenius automorphism). Thus, as for the $m$-sequence, both the 1-SSG and 0-SSG can be written using the absolute trace function.

## 2.1. Period of the 0-SSG

Here we shall find the period of the 0-SSG, using a similar approach as in [17].

**Theorem 1.** *Let $s$ be an $m$-sequence of length $2^n-1$, generated by a polynomial of degree $n$, with the respective 1-SSG and 0-SSG sequences extracted from it. The least period of the 1-SSG is a divisor of $2^{n-1}$, and the least period of the 0-SSG is a divisor of $2^{n-1}-1$. Moreover, the respective weights of both the 1-SSG and 0-SSG that are generated by $s$ are exactly $2^{n-2}$ (making the 1-SSG balanced, and 0-SSG almost balanced).*

*Proof.* Meier and Staffelbach [17] showed the result for the 1-SSG. We will now argue our claim for the 0-SSG. In any minimal cycle of an $m$-sequence, the 2-bit blocks $11, 10, 01$ occur $2^{n-2}$ times and 00 occurs $2^{n-2}-1$ times. Thus, the 0-SSG of length $2^{n-1}-1$ will have Hamming weight (that is, the number of nonzero bits) exactly $2^{n-2}$. Since 00 appears $2^{n-2}-1$ times and 01 occurs $2^{n-2}$ times in every minimal cycle of the $m$-sequence, then $2^{n-1}-1$ must be a period of the 0-SSG, and hence the least period must be a divisor of $2^{n-1}-1$.

## 2.2. Linear complexity of the 0-SSG

In this subsection, we shall find the linear complexity of the 0-SSG. The method used by Blackburn [1] cannot be applied directly to the 0-SSG as that method requires at least the sequence to have a power of 2 length, and our 0-SSG, say $z$, has length $2^{n-1}-1$. However, we can circumvent that by adding a single arbitrary bit (which we take to be 0, so that the new sequence is balanced) to a cycle (not necessary the minimal cycle) of the 0-SSG, and consequently, obtaining a sequence, say $\tilde{z}$, of length $2^{n-1}$. We claim that a period of $\tilde{z}$ is now $2^{n-1}$, and give the argument below. First, observe that if a cycle of $z$ is $z_0, z_1, \ldots, z_{2^{n-1}-2}$, and so

$$z = z_0, z_1, \ldots, z_{2^{n-1}-2}, z_0, z_1, \ldots, z_{2^{n-1}-2}, \ldots,$$

then

$$\tilde{z} = z_0, z_1, \ldots, z_{2^{n-1}-2}, 0, z_0, z_1, \ldots, z_{2^{n-1}-2}, 0, \ldots.$$

We note that the (balanced) string $z_0, z_1, \ldots, z_{2^{n-1}-2}, 0$ is of length $2^{n-1}$ and becomes a cycle of $\tilde{z}$. In spite of this modification, we still cannot adapt the proof from Blackburn [1], since, in fact, the bound $2^{n-1}-(n-2)$ will not hold for the 0-SSG, as we will see in our next theorem.

**Theorem 2.** *Let $n \geq 3$, and $s$ be an $m$-sequence of length $2^n-1$, generated by a primitive polynomial of degree $n$, along with the associated 1-SSG and 0-SSG sequences. The linear complexity of the 1-SSG is at most $2^{n-1}-(n-2)$ and of the 0-SSG is at most $2^{n-1}-2$, both bounds being attainable. A lower bound for both linear complexities is $2^{\lfloor n/2 \rfloor -1}$.*

*Proof.* The upper bound result on the 1-SSG was shown by Blackburn [1]. The attainability of the bounds can be seen from Table 2. The lower bound for the 1-SSG was shown by Meier-Staffelbach [17], and the one for the 0-SSG $z$ can be shown in the same manner. Let $\tilde{z}$ be the 0-SSG, with 0 inserted at the end of a cycle. The same argument of [17] shows that $LC(\tilde{z}) > 2^{\lfloor n/2 \rfloor -1}$. If $z$ can be generated by a recurrence of degree smaller than $2^{\lfloor n/2 \rfloor -1}$, then it can be generated by a recurrence of order at most $2^{\lfloor n/2 \rfloor -1}-1$ (it has to be odd, since we are working in binary). Then $\tilde{z}$ can be generated by a recurrence of order one more, namely, at most of order $2^{\lfloor n/2 \rfloor -1}$, which is not possible.

We now concentrate on the upper bound on the complexity. We showed earlier that the weight of the cycle $z_0, z_1, \ldots, z_{2^{n-1}-2}$ is exactly $2^{n-2}$ and an even number. Therefore, the associated polynomial $Z(x) = z_0 + z_1 x + \cdots + z_{2^{n-1}-2} x^{2^{n-1}-2}$ has an even number

of nonzero coefficients, and since we operate in binary, it will have $(x+1)$ as a factor. Thus, $\deg(\gcd(Z(x), 1-x^{2^{n-1}-2})) \geq 1$, and consequently, $LC(z) \leq 2^{n-1}-2$, which, as we observed, is attainable.

We will see later on in Table 4 that this upper bound is attained, for example, for $n = 3, 5, 6, 8, 9, 14$. We wondered whether the modification $\tilde{z}$ has a better linear complexity. It is known [14] that, for example, inserting a bit in the minimal cycle of $z$ of length $T$, generating $\tilde{z}$ of linear complexity

$$LC(\tilde{z}) \geq \min\left(LC(z), T+1-LC(z)\right).$$

In our case, $T = 2^{n-1}-1$, and so, $LC(\tilde{z}) \geq \min\left(LC(z), 2^{n-1}-LC(z)\right)$. As we see from Table 2, there are examples, $n \geq 5$, where the linear complexity of $z$ is maximal, namely, $LC(z) = 2^{n-1}-2$, and so, $LC(\tilde{z}) \geq \min\left(LC(z), 2^{n-1}-LC(z)\right) = 2$, which is not a useful bound.

## 3. Proposed Conflation Approach

Section 1 discussed the various weaknesses in sequences created by a shrinking generator. Our approach to addressing some of these weaknesses relies on an observation from Section 1 that what is discarded by the shrinking generator of the SSG is as good as what was kept. It relies on the fact that although the original SSG algorithm (1-SSG) chose to discard all bits in the form 00,01 and $10 \to 0$, $11 \to 1$, the opposite, discarding all bits in the form 10,11 and $00 \to 0$, $01 \to 1$ is equally as valid, and in some cases better as demonstrated in our discussion regarding the 0-SSG in Section 2. Our approach generates both the 1-SSG sequence and the 0-SSG sequence and XORs them to produce the final output sequence. The full algorithm is shown below. Also, while the data used in this paper is derived from LFSRs, other sequences can be used to drive the shrinking process.

One important consideration must be taken into account. The intermediate bitstreams $s_{shrink0}$ and $s_{shrink1}$ may not be of the same length as the number of 1's and 0's in the original bitstream is unlikely to be exactly equal. The algorithm uses the minimum length of either sequence to fix this problem.

In practice, this implementation would not be used. Instead of generating the full input bitstream $(a)$, we would generate one bit per clock cycle. We would then proceed by looking at pairs of bits and following the logic from Algorithm 1. The ending condition would be the number of bits desired. We used our algorithm instead of the practical implementation to generate the whole SCCG bit sequence in order to more accurately and easily perform the various statistical tests and evaluations. A hardware implementation of this more practical algorithm is left for future work.

### 3.1. Period of SSCG

**Theorem 3.** *Let $n$ be an integer with $n > 2$. The period of the SSCG is upper bounded by $2^{n-1} \cdot (2^{n-1}-1)$.*

**Algorithm 1:** Combining two shrunken bitstreams via XOR

**ShrinkBitstream** $(a)$
- **inputs :** Bitstream $a$
- **output:** Bistream *out*
- $out \leftarrow \emptyset$
- $s_{shrink0} \leftarrow \emptyset$
- $s_{shrink1} \leftarrow \emptyset$
- **foreach** pair of bits $(a_0, a_1)$ *in* $a$ **do**
  - **if** $(a_0, a_1) = (1, 0)$ **then**
    - Append 0 to $s_{shrink1}$;
  - **else if** $(a_0, a_1) = (1, 1)$ **then**
    - Append 1 to $s_{shrink1}$;
  - **else if** $(a_0, a_1) = (0, 0)$ **then**
    - Append 0 to $s_{shrink0}$;
  - **else if** $(a_0, a_1) = (0, 1)$ **then**
    - Append 1 to $s_{shrink0}$;
- $ind \leftarrow min(len(s_{shrink0}), len(s_{shrink1}))$
- **for** $i \leftarrow 0$ *to* $ind$ **do**
  - Append $s_{shrink0}[i] \oplus s_{shrink1}[i]$ to *out*
- **return** *out*;

*Proof.* We rely on three facts to show Theorem 3. In [1], an upper bound for the period of the 1-SSG is proven to be $2^{n-1}$. An upper bound for period of the 0-SSG found in Section 2.1 is $2^{n-1} - 1$. From [7, 20], the period of the XOR of two sequences is the product of the periods of the two streams if the periods are coprime. In our case, the 1-SSG and 0-SSG sequences have periods $2^{n-1}$ and $2^{n-1} - 1$, respectively, which are obviously coprime, and so, the maximum period of the SSCG is $2^{n-1} \cdot (2^{n-1} - 1)$.

**Remark 4.** *In fact, the proof shows that the period of SSCG is exactly* $per(0 - SSG) \cdot per(1 - SSG)$*, where* $per(s)$ *is the least period of the sequence* $s$*.*

### 3.2. Linear complexity of SSCG

**Theorem 5.** *The linear complexity of the SSCG is* $LC(SSCG) \leq 2^n - n$*.*

*Proof.* We rely on two facts to show Theorem 5. In [1], the maximal linear complexity of the 1-SSG is proven to be $2^{n-1} - (n - 2)$. We also use the fact [7, 20] that the linear complexity of the XOR of two sequences is the sum of the linear complexities of the two streams. In our case, our sequences have linear complexity $2^{n-1} - (n - 2)$ and $2^{n-1} - 2$ respectively, therefore, $LC(SSCG) \leq 2^n - n$.

## 4. Cryptanalysis

In this section we consider some of the possible attacks against the SSCG based on the attacks used to break the SSG. Despite the close relationship between the SSCG and the SSG, some of the attack approaches used against the SSG are not as applicable to the SSCG due to the XOR addition of the 0-SSG and 1-SSG streams.

### 4.1. Guess-and-Determine Attacks

There are several variants of guess-and-determine attacks against the SSG. Some examples of such works are [6], [10], [18], [19], and [25]. Though their approaches are slightly different, they all depend on an observation most succinctly stated in [25], that given the decimated sequence $a_{2i}$, it is possible to determine the original sequence $a_i$. In the case of the SSCG, with knowledge of either the 1-SSG or the 0-SSG, it would be possible to determine the original sequence. However, we do not see a method to go from our keystream, which is an XOR-masked (conflated) sum of the 0-SSG and 1-SSG sequences, and recover the underlying 1-SSG sequence required in order to carry out these aforementioned attacks. We will be more precise here to convince the reader. Given an $m$-sequence $\{a_i\}_i$ of feedback polynomial $f$ of degree $L$, let $h(x)=\sum_{j=0}^{L-1} h_j x^j$ be a polynomial such that $h(x)\equiv x^\tau \pmod{f^*(x)}$, where $\tau=2^{L-1}$ is the shift value between the even/odd subsequences $\{a_{2i}\}_i, \{a_{2i+1}\}_i$. The idea of the attack [25] (which improves upon Mihaljević's attack [18]) is the following. Since the even and odd indexed subsequences are shifts of the original $\{a_i\}_i$, then, if $a_{2i}=1$, and guessing $\ell$ (large but fixed number of) bits in the odd-indexed subsequence, then

$$b_i=a_{2i+1}=\sum_{j=1}^{\ell-1} h_j a_{2(i+j)}+\sum_{j=\ell}^{L-1} h_j a_{2(i+j)}=z_{\sum_{j=0}^{i-1} a_{2j}},$$

the output of the 1-SSG (we note a typo in [25] in the last sum). Thus, we get a linear equation in the $L-\ell$ unknown bits, guessing the first $\ell$ bits. The key (and used) observation of [25] is that the more ones in the guessed segment of length $\ell$ the more linear equations in the remaining $L-\ell$ bits one gets. Redoing the argument of [25], we can also find that the outputs of the 0-SSG, say $w_s$, can be represented as a combination of the even indexed terms, obtaining, for $a_{2i}=0$

$$b_k=a_{2k+1}=\sum_{j=1}^{\ell-1} h_j a_{2(k+j)}+\sum_{j=\ell}^{L-1} h_j a_{2(k+j)}=w_{k-\sum_{j=0}^{k-1} a_{2j}}.$$

However, the difficulty is the buffering, as the outputs of the 0-SSG and 1-SSG occur at different clocks and we see no way of finding the indices $i,k$ such that $\sum_{j=0}^{i-1} a_{2j}=k-\sum_{j=0}^{k-1} a_{2j}$, making a known ciphertext attack unsuccessful.

### 4.2. Exhaustive search and entropy attacks

The two most basic methods to reconstruct the initial state of the LFSR from a given SSG output sequence were proposed in [17]. Their proposed attacks made use of short keystreams requiring $O(2^{.79n})$ and $O(2^{.75n})$ computational steps respectively [17]. As our approach differs from those proposed in [3, 13, 17], our analysis with regards to exhaustive search is slightly different. Let $(b_0, b_1, b_2, \ldots, b_n)$ be a portion of the keystream generated by the SSCG. The bit $b_k$, for some $k$ where $0 \leq k \leq n$, can be generated by the 4-tuple $(a_i, a_{i+1}, a_{i+2}, a_{i+3})$, where the sequence $a$ is the output from the chosen LFSR and the index $i$ is unknown. The bit $b_k$ can be either 0 or 1 and the analysis for either case provides us the difficulty of using these two approaches. The analysis for an arbitrary bit, $b_k = 1$ is presented below.

| 4-tuple | Probability |
|---|---|
| 0000 | 1/16 |
| 0001 | 1/16 |
| 0010 | 0 |
| 0011 | 1/8 |
| 0100 | 1/16 |
| 0101 | 1/16 |
| 0110 | 1/8 |
| 0111 | 0 |
| 1000 | 0 |
| 1001 | 1/8 |
| 1010 | 1/16 |
| 1011 | 1/16 |
| 1100 | 1/8 |
| 1101 | 0 |
| 1110 | 1/16 |
| 1111 | 1/16 |

Table 3: Possible 4-tuples for bit output

As we know the value of $b_k$ to be 1, we know that certain bit tuples will give us a value of 0, thus they have probability of 0. The remaining 4-tuples have the values as specified. It is important to note that this scenario is the best case scenario for the attacker in which every 4-tuple is independent from the next. In the original SSG, bit pairs of the form 00 or 01 were discarded. Our approach does not discard these values, rather we store them for use. Thus, our worst case (from the defender perspective) for the number of states, given that we need to reconstruct $r$ 4-tuples and there are $q = 4r$ total bits is: $12^{\frac{q}{4}} = 2^{(\log_2 12)\frac{q}{4}} = 2^{1.861q}$. Again, we point out that this analysis cannot be too accurate but provides a lower bound for the exhaustive search complexity.

The total block entropy can also be determined using the probability values from

Table 3 as

$$H = -\left(\frac{8}{16}\log_2\frac{1}{16} + \frac{4}{8}\log_2\frac{1}{8}\right) = \frac{7}{2}.$$

The entropy per bit is $\frac{7}{8}$. This means that an entropy search of all $q$ bits would require $2^{.875q}$. Our approach is more secure than the SSG and its variants proposed in [3, 17]. This approach is also imperfect in that a block that results in a 0 or 1 need not be of length 4 but presents a bound on the difficulty of the process.

## 5. Validation Experiments

The software used in this research was run on laptop running Ubuntu 16.04 with 16 GB of RAM, written in the Python3 programming language. Table 4 presents both period and linear complexity results for a polynomial with degree $n$ for the SSCG, as well a comparison of those same properties with the SSG. Figure 3 shows a linear complexity profile comparison for the SSG and SSCG using the primitive polynomial $x^{10} + x^3 + 1$.

| n | SSG Period | SSCG Period | SSG LC | SSCG LC | Theor. Max LC |
|---|---|---|---|---|---|
| 3 | 4 | 12 | 3 | 5 | 5 |
| 4 | 8 | 56 | 5 | 8 | 12 |
| 5 | 16 | 240 | 13 | 27 | 27 |
| 6 | 32 | 992 | 28 | 58 | 58 |
| 7 | 64 | 4032 | 59 | 119 | 121 |
| 8 | 128 | 16256 | 122 | 248 | 248 |
| 9 | 256 | 65280 | 249 | 503 | 503 |
| 10 | 512 | 261632 | 504 | 1011 | 1014 |
| 11 | 1024 | 1047552 | 1015 | 2035 | 2037 |
| 12 | 2048 | 4192256 | 2038 | 4072 | 4084 |
| 13 | 4096 | 16773120 | 4085 | 8175 | 8179 |
| 14 | 8192 | 67100672 | 8180 | 16370 | 16370 |
| 15 | 16384 | 268419072 | 16371 | 32571 | 32753 |
| 16 | 32768 | 1073709056 | 32742 | 65519 | 65520 |

Table 4: Comparison of the period and linear complexity of the SSG and the SSCG

The period data for these polynomials matches what we would have expected from the formula that was presented in Section 3.1. Our values for linear complexity are very close (or reach) our proposed theoretical maximum linear complexity from Section 3.2. The SSCG outperforms the SSG in terms of both period and linear complexity. The linear complexity of the SSCG is approximately double than of the SSG and the period is several times greater. We also ran bitstreams produced by the SSCG against the statistical test suite provided by NIST (NIST SP 800-22) to evaluate the baseline security of pseudorandom number generators for cryptographic applications. The full description of the testing methodology can be found in [12]. We ran several SSCG bitstreams through the

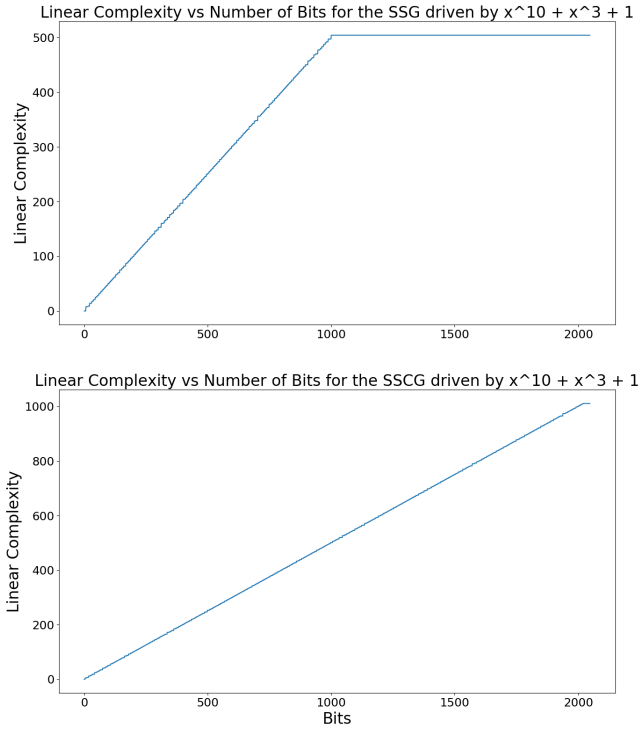


Figure 3: LC Comparison between SSG and SSCG for $x^{10} + x^3 + 1$

NIST suite and were able to pass every test. We have provided output of testing on the bitstream produced by the polynomial $x^{16}+x^{12}+x^3+x+1$ in the Appendix. Due to space considerations in the appendix, repeated tests were averaged. We ran the tests against 100 streams of 5 million bits each for analysis for that polynomial. The SSCG bitstreams passed every test from the suite.

Finally, we make some remarks on the amount of buffer bits required for our approach. Empirically, we have determined that the amount of buffer bits required is approximately $2^{n-1}$ bits where $n$ is the degree of the polynomial. For example, for the polynomial $x^{12} + x^9 + x^3 + x^2 + 1$ with seed 15, the SSCG period was 4192256 and the amount of buffer bits required was 1044. This is an expected result as when an $m$-sequence is repeated twice, the count of the bit pair 00, is one less than the other bit pairs. Over the course of the SSCG period, which requires the original $m$-sequence to be repeated $2^n$ times, we would expect a difference of $2^{n-1}$ in the count of 00. Compared to the period of the SSCG sequence, the buffer amount is rather small.

## 6. Conclusions

We have presented a computationally inexpensive modification to the Self-Shrinking Generator that exhibits improved period, linear complexity, and added security against several known SSG attacks. In the process of doing so, we examined the mathematical

properties of the 0-SSG, which to this point had not been explored. An analysis of the period and LC of the 0-SSG in turn allowed us to prove features of the SSCG. In conjunction with proofs for period and LC, we also presented data where our approach for the SSCG was implemented. That data matched what we had expected. Finally, we began a security evaluation of the SSCG by presenting bounds on some basic attacks and by running bitstreams produced by the SSCG against the NIST standards. While these positive results are not fully sufficient to put in practice the SSCG, they are valuable data points for a future, more thorough security evaluation.

There is work that remains to be done. As noted earlier, some classes of attacks that were envisioned against the SSG are not applicable or practical for the SSCG, but there are surely other attacks that can be leveraged against the SSCG. Furthermore, an efficient hardware implementation of the SSCG is a necessary step for more widespread use. There are several other interesting features of the SSCG that could be explored in the future. For example, it is interesting to note that while it is known [20, Proposition 4.6] that a random sequence of period $2^{n-1}$ has a linear complexity greater than $2^{n-1} - 1$, the 1-SSG is not respecting that and the 0-SSG $z$ is closer to respecting the random approach (sure, we work with the modified $\tilde{z}$ so that the period is exactly $2^{n-1}$). Furthermore, one can extend our approach to the odd characteristic, as well. There are many other such observations that could be the subject of future research.

## Acknowledgements

The authors thank the editor for quickly and efficiently handling our manuscript, and the referee for useful comments. The second and third-named authors thank the NPS Foundation for the grant support.

## References

[1] S. R. Blackburn. The linear complexity of the self-shrinking generator. *IEEE Transactions on Information Theory*, 45(6):2073–2077, September 1999.

[2] S. D. Cardell and A. Fúster-Sabater. The t-modified self-shrinking generator. In Y. Shi, H. Fu, Y. Tian, V. V. Krzhizhanovskaya, M. H. Lees, J. Dongarra, and P. M. A Sloot, editors, *Computational Science – ICCS 2018*, pages 653–663, Cham, 2018. Springer International Publishing.

[3] K.-Y. Chang, J.-S. Kang, M.-K. Lee, H. Lee, and D. Hong. New variant of the self-shrinking generator and its cryptographic properties. In M.-S. Rhee and B. Lee, editors, *Information Security and Cryptology – ICISC 2006*, pages 41–50, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] D. Coppersmith, H. Krawczyk, and Y. Mansour. The shrinking generator. In D. R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 22–39, 1994.

[5] T. W. Cusick and P. Stănică. *Cryptographic Boolean Functions and Applications (2nd ed.)*. Elsevier-Academic Press, 2017.

[6] B. Debraize and L. Goubin. Guess-and-determine algebraic attack on the self-shrinking generator. In *Proceedings of the International Workshop on Fast Software Encryption*, volume 5086, pages 235–252. Springer-Verlag, 2008.

[7] S. W. Golomb. *Shift register sequences: secure and limited-access code generators, efficiency code generators, prescribed property generators, mathematical models*. World Scientific, New Jersey, third revised edition, 2017.

[8] M. Goresky and A. Klapper. *Algebraic shift register sequences*. Cambridge University Press, 2012.

[9] M. Hazewinkel, editor. *Handbook of algebra*. Elsevier, Amsterdam; New York, 1996.

[10] M. Hell and T. Johansson. Two new attacks on the self-shrinking generator. *IEEE Transactions on Information Theory*, 52(8):3837–3843, 2006.

[11] Y. Hu and G. Xiao. Generalized Self-Shrinking Generator. *IEEE Transactions on Information Theory*, 50(4):714–719, April 2004.

[12] L.E. Bassham III, 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*. National Institute of Standards & Technology, 2010.

[13] A. Kanso. Modified self-shrinking generator. *Computers & Electrical Engineering*, 36(5):993–1001, September 2010.

[14] R. Kavuluru and A. Klapper. Lower bounds on error complexity measures for periodic lfsr and fcsr sequences. *Cryptography and Communications*, 1(1):95–116, 2009.

[15] M. Krause. Bdd-based cryptanalysis of keystream generators. In L. R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 222–237, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[16] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2nd edition, 1997.

[17] W. Meier and O. Staffelbach. The self-shrinking generator. In A. De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 205–214, 1995.

[18] M. J. Mihaljević. A faster cryptanalysis of the self-shrinking generator. In J. Pieprzyk and J. Seberry, editors, *Information Security and Privacy*, pages 182–189, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[19] M.-E. Pazo-Robles and A. Fúster-Sabater. Cryptanalytic attack on the self-shrinking sequence generator. In *Proceedings of the 10th International Conference on Adaptive and Natural Computing Algorithms - Volume Part II*, ICANNGA'11, page 285–294, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] R. A. Rueppel. *Analysis and design of stream ciphers.* Springer-Verlag, 1986.

[21] B. Subba, S. Biswas, and S. Karmakar. Intrusion detection in Mobile Ad-hoc Networks: Bayesian game formulation. *Engineering Science and Technology, an International Journal*, 19(2):782–799, June 2016.

[22] J. R. Vacca, editor. *Computer and information security handbook.* Morgan Kaufmann Publishers, an imprint of Elsevier, Cambridge, MA, third edition, 2017.

[23] S. Wolfram. Solomon Golomb (1932–2016)—Stephen Wolfram Writings.

[24] E. Zenner, M. Krause, and S. Lucks. Improved cryptanalysis of the self-shrinking generator. In V. Varadharajan and Y. Mu, editors, *Information Security and Privacy*, pages 21–35, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[25] B. Zhang and D. Feng. New guess-and-determine attack on the self-shrinking generator. In *Proceedings of the 12th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'06, pages 54–68, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] B. Zhang, H. Wu, D. Feng, and F. Bao. Security analysis of the generalized self-shrinking generator. In J. Lopez, S. Qing, and E. Okamoto, editors, *Information and Communications Security*, pages 388–400, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[27] K. Zrar. *Smart cities cybersecurity and privacy.* Elsevier, Cambridge, MA, first edition, 2018.

# Appendix A: Linear complexity data for 0-SSG versus 1-SSG for a variety of polynomials

| Polynomial | 1-SSG LC | 0-SSG LC | Polynomial | 1-SSG LC | 0-SSG LC |
|---|---|---|---|---|---|
| $x^2+x+1$ | 2 | 1 | $x^3+x+1$ | 3 | 2 |
| $x^4+x+1$ | 5 | 3 | $x^5+x^2+1$ | 12 | 12 |
| $x^5+x^4+x^2+x+1$ | 13 | 14 | $x^5+x^4+x^3+x^2+1$ | 10 | 12 |
| $x^6+x+1$ | 28 | 25 | $x^6+x^5+x^2+x+1$ | 26 | 30 |
| $x^6+x^5+x^3+x^2+1$ | 28 | 30 | $x^7+x+1$ | 57 | 62 |
| $x^7+x^3+x^2+x+1$ | 58 | 60 | $x^7+x^3+1$ | 59 | 62 |
| $x^7+x^5+x^4+x^3+x^2+x+1$ | 59 | 56 | $x^7+x^4+x^3+x^2+1$ | 54 | 59 |
| $x^7+x^6+x^4+x^2+1$ | 58 | 57 | $x^7+x^6+x^3+x+1$ | 58 | 57 |
| $x^7+x^6+x^5+x^4+x^2+x+1$ | 56 | 62 | $x^7+x^6+x^5+x^2+1$ | 59 | 60 |
| $x^8+x^5+x^3+x+1$ | 121 | 119 | $x^8+x^4+x^3+x^2+1$ | 121 | 126 |
| $x^8+x^6+x^5+x+1$ | 119 | 126 | $x^8+x^6+x^4+x^3+x^2+x+1$ | 122 | 119 |
| $x^8+x^6+x^5+x^3+1$ | 122 | 119 | $x^8+x^6+x^5+x^2+1$ | 122 | 126 |
| $x^8+x^7+x^6+x^5+x^2+x+1$ | 122 | 126 | $x^8+x^7+x^6+x+1$ | 122 | 126 |
| $x^9+x^5+x^3+x^2+1$ | 245 | 250 | $x^9+x^4+1$ | 249 | 254 |
| $x^9+x^6+x^5+x^3+x^2+x+1$ | 249 | 236 | $x^9+x^6+x^4+x^3+1$ | 249 | 242 |
| $x^9+x^7+x^6+x^4+x^3+x+1$ | 249 | 254 | $x^9+x^6+x^5+x^4+x^2+x+1$ | 249 | 252 |
| $x^9+x^8+x^5+x^4+1$ | 249 | 254 | $x^9+x^8+x^4+x+1$ | 246 | 254 |
| $x^9+x^8+x^6+x^5+x^3+x+1$ | 249 | 254 | $x^9+x^8+x^6+x^5+1$ | 249 | 242 |
| $x^9+x^8+x^7+x^3+x^2+x+1$ | 249 | 250 | $x^9+x^8+x^7+x^2+1$ | 247 | 254 |
| $x^9+x^8+x^7+x^6+x^5+x^3+1$ | 247 | 250 | $x^9+x^8+x^7+x^6+x^5+x+1$ | 243 | 252 |
| $x^{10}+x^4+x^3+x+1$ | 504 | 501 | $x^{10}+x^3+1$ | 504 | 507 |
| $x^{10}+x^8+x^3+x^2+1$ | 504 | 501 | $x^{10}+x^6+x^5+x^3+x^2+x+1$ | 504 | 507 |
| $x^{10}+x^8+x^5+x+1$ | 504 | 510 | $x^{10}+x^8+x^4+x^3+1$ | 504 | 510 |
| $x^{10}+x^8+x^7+x^6+x^5+x^2+1$ | 504 | 507 | $x^{10}+x^8+x^5+x^4+1$ | 504 | 510 |
| $x^{10}+x^9+x^4+x+1$ | 504 | 510 | $x^{10}+x^8+x^7+x^6+x^5+x^4+x^3+x+1$ | 504 | 507 |
| $x^{10}+x^9+x^8+x^6+x^3+x^2+1$ | 500 | 510 | $x^{10}+x^9+x^6+x^5+x^4+x^3+x^2+x+1$ | 502 | 510 |
| $x^{10}+x^9+x^8+x^7+x^6+x^5+x^4+x^3+1$ | 504 | 507 | $x^{10}+x^9+x^8+x^6+x^5+x+1$ | 498 | 510 |
| $x^{11}+x^5+x^3+x+1$ | 1013 | 1022 | $x^{11}+x^2+1$ | 1015 | 1017 |
| $x^{11}+x^6+x^5+x+1$ | 1011 | 1007 | $x^{11}+x^5+x^3+x^2+1$ | 1014 | 1022 |
| $x^{11}+x^8+x^5+x^2+1$ | 1015 | 1020 | $x^{11}+x^7+x^3+x^2+1$ | 1014 | 1020 |
| $x^{11}+x^8+x^6+x^5+x^4+x^3+x^2+x+1$ | 1015 | 1022 | $x^{11}+x^8+x^6+x^5+x^4+x+1$ | 1013 | 1010 |
| $x^{11}+x^9+x^8+x^7+x^4+x+1$ | 1013 | 1020 | $x^{11}+x^9+x^4+x+1$ | 1012 | 1022 |
| $x^{11}+x^{10}+x^7+x^4+x^3+x+1$ | 1015 | 1022 | $x^{11}+x^{10}+x^3+x^2+1$ | 1014 | 1022 |
| $x^{11}+x^{10}+x^9+x^8+x^3+x+1$ | 1012 | 1022 | $x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^3+x+1$ | 1014 | 1012 |
| $x^{12}+x^9+x^3+x^2+1$ | 2038 | 2035 | $x^{12}+x^6+x^4+x+1$ | 2037 | 2035 |
| $x^{12}+x^{10}+x^9+x^8+x^6+x^2+1$ | 2037 | 2046 | $x^{12}+x^9+x^8+x^3+x^2+x+1$ | 2038 | 2046 |
| $x^{12}+x^{11}+x^6+x^4+x^2+x+1$ | 2037 | 2046 | $x^{12}+x^{10}+x^9+x^8+x^6+x^5+x^4+x^2+1$ | 2038 | 2046 |
| $x^{12}+x^{11}+x^9+x^7+x^6+x^4+1$ | 2038 | 2035 | $x^{12}+x^{11}+x^9+x^5+x^3+x+1$ | 2037 | 2046 |
| $x^{12}+x^{11}+x^9+x^8+x^7+x^4+1$ | 2038 | 2046 | $x^{12}+x^{11}+x^9+x^7+x^6+x^5+1$ | 2038 | 2046 |
| $x^{12}+x^{11}+x^{10}+x^5+x^2+x+1$ | 2037 | 2046 | $x^{12}+x^{11}+x^9+x^8+x^7+x^5+x^2+x+1$ | 2037 | 2035 |
| $x^{12}+x^{11}+x^{10}+x^9+x^8+x^7+x^5+x^4+x^3+x+1$ | 2036 | 2046 | $x^{12}+x^{11}+x^{10}+x^8+x^6+x^4+x^3+x+1$ | 2037 | 2046 |
| $x^{13}+x^9+x^7+x^5+x^4+x^3+x^2+x+1$ | 4084 | 4090 | $x^{13}+x^4+x^3+x+1$ | 4083 | 4092 |
| $x^{13}+x^{10}+x^9+x^7+x^5+x^4+1$ | 4085 | 4094 | $x^{13}+x^9+x^8+x^7+x^5+x+1$ | 4084 | 4094 |
| $x^{13}+x^{11}+x^8+x^7+x^4+x+1$ | 4083 | 4091 | $x^{13}+x^{10}+x^9+x^8+x^6+x^3+x^2+x+1$ | 4085 | 4089 |
| $x^{13}+x^{12}+x^6+x^5+x^4+x^3+1$ | 4084 | 4094 | $x^{13}+x^{11}+x^{10}+x^9+x^8+x^7+x^6$ $+x^5+x^4+x^3+x^2+x+1$ | 4085 | 4082 |
| $x^{13}+x^{12}+x^9+x^8+x^4+x^2+1$ | 4085 | 4091 | $x^{13}+x^{12}+x^8+x^7+x^6+x^5+1$ | 4085 | 4094 |
| $x^{13}+x^{12}+x^{11}+x^5+x^2+x+1$ | 4079 | 4094 | $x^{13}+x^{12}+x^{10}+x^8+x^6+x^4+x^3+x^2+1$ | 4085 | 4094 |
| $x^{13}+x^{12}+x^{11}+x^9+x^5+x^3+1$ | 4084 | 4091 | $x^{13}+x^{12}+x^{11}+x^8+x^7+x^6+x^4+x+1$ | 4085 | 4092 |
| $x^{14}+x^{10}+x^6+x+1$ | 8180 | 8190 | $x^{14}+x^8+x^6+x+1$ | 8180 | 8190 |
| $x^{14}+x^{11}+x^6+x+1$ | 8180 | 8177 | $x^{14}+x^{10}+x^9+x^7+x^6+x^4+x^3+x+1$ | 8178 | 8190 |
| $x^{14}+x^{12}+x^9+x^8+x^7+x^6+x^5+x^4+1$ | 8180 | 8177 | $x^{14}+x^{11}+x^9+x^6+x^5+x^2+1$ | 8180 | 8190 |
| $x^{14}+x^{12}+x^{11}+x^{10}+x^9+x^7+x^4+x^3+1$ | 8180 | 8190 | $x^{14}+x^{12}+x^{11}+x^9+x^8+x^7+x^6+x^5+x^3+x+1$ | 8179 | 8190 |
| $x^{14}+x^{13}+x^{10}+x^8+x^7+x^5+x^4+x^3+x^2+x+1$ | 8178 | 8190 | $x^{14}+x^{13}+x^6+x^5+x^3+x+1$ | 8179 | 8190 |
| $x^{14}+x^{13}+x^{11}+x^8+x^5+x^3+x^2+x+1$ | 8179 | 8190 | $x^{14}+x^{13}+x^{11}+x^6+x^5+x^4+x^2+x+1$ | 8178 | 8190 |
| $x^{14}+x^{13}+x^{12}+x^{11}+x^{10}+x^9+x^6+x^5+1$ | 8179 | 8177 | $x^{14}+x^{13}+x^{12}+x^{11}+x^{10}+x^7+x^6+x+1$ | 8179 | 8190 |
| $x^{15}+x^4+1$ | 16371 | 16382 | $x^{15}+x+1$ | 16369 | 16382 |
| $x^{15}+x^7+x^6+x^3+x^2+x+1$ | 16365 | 16382 | $x^{15}+x^7+1$ | 16371 | 16380 |
| $x^{15}+x^{10}+x^5+x^4+1$ | 16367 | 16382 | $x^{15}+x^{10}+x^5+x+1$ | 16371 | 16375 |
| $x^{15}+x^{10}+x^9+x^7+x^5+x^3+1$ | 16371 | 16382 | $x^{15}+x^{10}+x^5+x^4+x^2+x+1$ | 16371 | 16382 |
| $x^{15}+x^{11}+x^7+x^6+x^2+x+1$ | 16369 | 16380 | $x^{15}+x^{10}+x^9+x^8+x^5+x^3+1$ | 16371 | 16373 |
| $x^{15}+x^{12}+x^5+x^4+x^3+x^2+1$ | 16370 | 16382 | $x^{15}+x^{12}+x^3+x+1$ | 16368 | 16382 |
| $x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+x^{10}+x^9+$ $x^8+x^7+x^6+x^5+x^4+x^3+x^2+1$ | 16371 | 16382 | $x^{15}+x^{12}+x^{11}+x^8+x^7+x^6+x^4+x^2+1$ | 16371 | 16382 |

Table 5: Comparison of SSG approaches using either zeroes or ones for a sample of primitive polynomials up to degree 15

## Appendix B: Output of NIST test suite for 100 bitstreams of 5 million bits using the polynomial $x^{16} + x^{12} + x^3 + x + 1$

```
------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF
PASSING SEQUENCES
------------------------------------------------------------------------
   generator is <data/16_stream_full.txt>
------------------------------------------------------------------------
P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------
0.020548     98/100     Frequency
0.224821     99/100     BlockFrequency
0.001895     98/100     CumulativeSums
0.035174     98/100     CumulativeSums
0.037566     98/100     Runs
0.289667    100/100     LongestRun
0.224821     99/100     Rank
0.096578     97/100     FFT
0.511764     99/100     NonOverlappingTemplate*
0.437274     99/100     OverlappingTemplate
0.867692     98/100     Universal
0.275709     99/100     ApproximateEntropy
0.537564     88/89      RandomExcursions*
0.293242     88/89      RandomExcursionsVariant*
0.657933    100/100     Serial
0.574903     99/100     Serial
0.455937     97/100     LinearComplexity

*148 NonOverlappingTemplate tests were performed. P-Value and
Proportion shown are averages.
*8 RandomExcursions tests were performed. P-Value and Proportion
shown are averages.
*18 RandomExcursionsVariant tests were performed. P-Value and
Proportion shown are averages.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception
of the random excursion (variant) test is approximately = 96 for a
sample size = 100 binary sequences.
The minimum pass rate for the random excursion (variant) test
is approximately = 85 for a sample size = 89 binary sequences.
For further guidelines construct a probability table using the
MAPLE program provided in the addendum section of the documentation.
```