A Practical Cryptanalysis of SSC2

Philip Hawkes[1], Frank Quick[2], and Gregory G. Rose[1]

[1] Qualcomm Australia, Level 3, 230 Victoria Rd, Gladesville, NSW 2111, Australia
[2] Qualcomm Inc., 5775 Morehouse Drive, San Diego, CA 92121-1714, USA
{phawkes,fquick,ggr}@qualcomm.com

Abstract. SSC2 is a stream cipher that operates by XORing the output of two "half-ciphers". The first half-cipher is constructed from a linear feedback shift register (LFSR) with a non-linear filter. The second half-cipher is constructed from a lagged Fibonacci generator (LFG) and a multiplexor that chooses values from the Fibonacci register. The second half-cipher has a small cycle length $\pi \approx 2^{52}$. The initial state of the LFSR is derived by performing a fast correlation attack on the sequence resulting when XORing the key-stream at an interval of π words (thus cancelling the effect of the LFG). This attack requires around 2^{25} words of this sequence and a few hours of computation. The initial state of the LFG is then derived from around 15300 outputs using around one second of computation.

Keywords: SSC2, fast correlation attack.

1 Introduction

SSC2 is a stream cipher proposed by Zhang, Carroll and Chan [2]. The cipher is designed for software implementation and is very fast. This paper describes a practical cryptanalysis of SSC2 that requires around 2^{25} words of known key-stream (from a run of 2^{52} words) and a few hours work on a 250 MHz processor with 100 MB of memory.

SSC2 is based on a *linear feedback shift register* (LFSR) and a *lagged Fibonacci generator* (LFG). An LFSR consists of a register that stores a set of bits called the *state*, and a function that is linear modulo 2. This function updates the state bit-by-bit. An LFG consists of a register which stores a set of integers modulo N (once again called the state) and a function that is linear modulo N. This function updates the state integer-by-integer. In SSC2, the modulus is $N = 2^{32}$, and the integers are stored as 32-bit blocks called *words*.

SSC2 achieves its speed by using 32-bit operations. The stream is derived from a 127-bit LFSR, a 17-word LFG and a multiplexor that chooses values from the register of the LFG. The 127-bit register for the LFSR is stored in four 32-bit words (the extra bit is forced to 1 in the filter function). After the states of the LFSR and LFG are initialised, the following steps are repeated to produce each word of output:

S. Vaudenay and A. Youssef (Eds.): SAC 2001, LNCS 2259, pp. 25–37, 2001.

1. 32-bits of the LFSR state are updated simultaneously. A *non-linear filter* (NLF) computes a 32-bit output N_i from the four words in the state of the LFSR.
2. The LFG state is updated. The upper 16-bits and lower 16-bits of the LFG output are swapped to form L_i.
3. The multiplexor uses the four most significant bits (MSBs) of the updated word to choose one of 16 values in the LFG state to be the output M_i.
4. The output of the cipher is $Z_i = (L_i + M_i \bmod 2^{32}) \oplus N_i$, where $\oplus$ denotes XOR.

The value N_i is called the output of the LFSR half-cipher, while $V_i = (L_i + M_i \bmod 2^{32})$ is called the output of the LFG half-cipher.

Previous Results. In the rump session of Crypto 2000, Rose and Hawkes [6] reported on correlations between the least significant bits (LSBs) of certain words output from SSC2. They also noted that the LFG has a small period $\pi = 17 \cdot 2^{31} \cdot (2^{17} - 1) \approx 2^{52}$. Computing $Z'_i \stackrel{\text{def}}{=} Z_i \oplus Z_{i+\pi} = N_i \oplus N_{i+\pi}$, allows the LFSR to be attacked in isolation. The correlation in the LSBs of Z'_i allows an attacker to distinguish the output of SSC2 from a random bit stream. Another analysis by Hawkes and Rose [7] found an attack on the LFSR half-cipher in isolation that requires 382 words and around 2^{42} time. Bleichenbacher and Meier [1] found an attack on the entire cipher that finds the initial state of the LFSR using around 2^{52} words of known key-stream with around 2^{75} time. This attack exploits the small period π. Following this, the initial state of the LFG is found using around 2^{32} known outputs of the LFG half-cipher with around 2^{75} time.

Concurrent Results. Independently, Fluhrer, Crowley and Harvey had also identified a number of correlations in the LFSR half-cipher [4], and give other attacks. They noticed that there are actually two different correlations, apparently equally valid, with the LSB of the N_i.

New Results. The first part of the attack in this paper exploits the small period of the LFG by performing a fast correlation attack on the stream Z'_i, based on the correlation noted in [6]. This part of the attack requires around 2^{25} words of known key-stream (from a run of 2^{52} words) with a few hours of processing time on a 250 MHz Sun UltraSPARC (see Section 3). The attack applies simple techniques that increase the accuracy and speed of any fast correlation attack. After the output of the LFSR half-cipher is removed, the attack exploits properties of the LFG noted in [1] to identify when the multiplexor has selected specific words in the LFG register. This information is used to reconstruct the initial state of the LFG (Section 4). This part of the attack requires around 15300 known outputs of the LFG half-cipher (presumed already known from the previous phase) and around a second of processing on a 250 MHz Sun UltraSPARC.

2 A Description of SSC2

LFSR half-cipher. The LFSR state is stored as four 32-bit words denoted $(X_{i+3}, X_{i+2}, X_{i+1}, X_i)$. The state is updated to $(X_{i+4}, X_{i+3}, X_{i+2}, X_{i+1})$ by

computing

$$X_{i+4} = X_{i+2} \oplus (X_{i+1} << 31) \oplus (X_i >> 1),$$

where '<<' denotes a zero-fill left shift and '>>' denotes a zero-fill right shift. The least significant bit of X_i is ignored. If this sequence were converted to a bit-stream b_t, then the bit-sequence would satisfy the linear recursion:

$$b_{t+127} = b_{t+63} + b_t \bmod 2.$$

The corresponding characteristic polynomial is $x^{127} + x^{63} + 1$. This polynomial is irreducible modulo 2, which means that the bit sequence has a period of $(2^{127} - 1)$. The LFSR is implemented using a 4-word array $S[1], \ldots, S[4]$ containing $X_{i+3}, \ldots, X_i$. At each clock, the LFSR computes $A = S[2] \oplus (S[3] << 31) \oplus (S[4] >> 1)$. The values are shifted up ($S[4] \leftarrow S[3]$, $S[3] \leftarrow S[2]$, $S[2] \leftarrow S[1]$) and the value of $S[1]$ is set to A. After the LFSR is updated, the NLF output N_i is computed. The NLF uses a variety of operations: XOR; modular addition; $SWAP(A)$: swaps the upper 16-bits and lower 16-bits of A; and $\widehat{X}_i$: which denotes the word X_i with the LSB forced to 1.

NLF Algorithm

1 $A \leftarrow X_{i+3} + \widehat{X}_i \bmod 2^{32}$, with $c1 \leftarrow$ carry;
2 $A \leftarrow SWAP(A)$;
3 **if** $(c1 = 0)$ **then** $A \leftarrow X_{i+2} + A \mod 2^{32}$ with $c2 \leftarrow$ carry;
4 **else** $A \leftarrow (X_{i+2} \oplus \widehat{X}_i) + A \mod 2^{32}$ with $c2 \leftarrow$ carry;
5 $N_i \leftarrow (X_{i+1} \oplus X_{i+2}) + A + c2 \mod 2^{32}$;

The LFG half-cipher. The LFG state consists of 17 words $(Y_{i+16}, \ldots, Y_i)$. The state is updated to $(Y_{i+17}, \ldots, Y_{i+1})$ using the recurrence:

$$Y_{i+17} = Y_{i+12} + Y_i \bmod 2^{32}. \tag{1}$$

The LFG is implemented using a 17-word array $G[1], \ldots, G[17]$. The key scheduling initialises $G[1], \ldots, G[17]$ to the values $Y_{16}, \ldots, Y_0$, and initialises two pointers r and s to 17 and 5 respectively. The output L_i is defined as $L_i = SWAP(Y_i)$. The LFG state is updated by computing

$$G[r] + G[s] \;=\; Y_i + Y_{i+12} \;=\; Y_{i+17} \bmod 2^{32},$$

and replacing the value of $G[r]$ (which was Y_i) with the value of Y_{i+17}. The values of r and s are then decreased by 1 (when r or s reaches 0, the value is reset to 17). The output M_i is defined as

$$M_i \;=\; G[1 + (s + (Y_{i+17} >> 28) \bmod 16)].$$

As a result of the reduction modulo 16, the formula for M_i in terms of the sequence $\{Y_i\}$ changes according to the value of i mod 17. Now that L_i, M_i and N_i have been computed, SSC2 outputs $Z_i = ((L_i + M_i \bmod 2^{32}) \oplus N_i)$, increments i and repeats the process. This paper does not address the issue of obtaining the key from the initial states of the LFSR and LFG, so we do not describe the key scheduling algorithm.

3 Attacking the LFSR Half-Cipher

The attack on the LFSR half-cipher is an advanced fast correlation attack, exploiting an observed correlation between the least significant bit of the filtered output words and five of the LFSR state bits. The attack is aided greatly by the fact that the feedback polynomial of the LFSR is only a trinomial: $x^{127}+x^{63}+1$. Meier and Staffelbach observed in [10] in 1989 "any correlation to an LFSR with less than 10 taps should be avoided".

3.1 Background: Fast Correlation Attacks

The seminal work on Fast Correlation Attacks is [10], and another paper which explains them and explores some heuristic optimisations is [5].

Many stream ciphers have an underlying Linear Feedback Shift Register, and produce output by applying some nonlinear function to the state of the register; many schemes which appear different in structure are equivalent to this formulation. SSC2's LFSR half-cipher is such a construction.

If the nonlinear function is perfect, there should be no (useful) correlation between the output of the generator and any linear function of the state bits. Conversely, if there is a correlation between output bits and any linear combination of the state bits, this may be used by a fast correlation attack to recover the initial state. Consider the output bits of the generator, $\{B_i\}$, to be outputs from an LFSR, $\{A_i\}$, modified by erroneous bits $\{E_i\}$ with some probability $P < 0.5$. The probability of error P is the opposite of the known correlation. Put simply, the technique of a Fast Correlation Attack utilises the recurrence relations obeyed by the X_i to identify particular bits in the output stream which have a high probability of being erroneous, and correct (flip) them. To do this, the attack computes $(B_j + \sum_{i \in T} B_i \bmod 2)$, for each recurrence relation $A_j + \sum_{i \in T} A_i \equiv 0 \pmod 2$, (these are also called parity check equations). The error probability for bit j: $P(B_j \neq A_j)$, is computed based on the number of recurrence relations $(B_j + \sum_{i \in T} B_i \equiv 0 \bmod 2)$ satisfied and the number of recurrence relations unsatisfied. If there are enough bits in the output stream for the given P, this process will eventually converge until a consistent LFSR output stream remains. Linear algebra is then used to recover the corresponding initial state of the LFSR.

3.2 Fast Correlation Attack on SSC2

Recall that $\pi = 17 \cdot 2^{31} \cdot (2^{17} - 1)$ is the period of the Lagged Fibonacci Generator half-cipher. If two segments of output stream π apart are exclusive-ored together, the contributions from the LFG half-cipher cancel out, leaving the exclusive-or of two filtered LFSR streams to be analysed.

Let $Z'_i = Z_i \oplus Z_{i+\pi} = N_i \oplus N_{i+\pi}$. N_i exhibits a correlation to a linear function of the bits of the four-word state S_i. Define $l(S) = S[1]_{15} \oplus S[1]_{16} \oplus S[2]_{31} \oplus S[3]_0 \oplus S[4]_{16}$, where the subscript indicates a particular bit of the word (with bit 0 being the least significant bit). Then $P(\mathrm{LSB}(Z_i) = l(S_i)) = 5/8$. (Note that

this correlation is incorrectly presented in [1]). Intuitively, three of these terms are the bits that are XORed to form the least significant bits of N_i; the other two terms contribute to the carry bits that influence how this result might be inverted or affected by carry propagation. Obviously $N_{i+\pi}$ is similarly correlated to the state $S_{i+\pi}$, but because the state update function is entirely linear, the bits of $S_{i+\pi}$ are in turn linear functions of the bits of S_i. So $\mathrm{LSB}(Z'_i)$ exhibits a correlation to $L(S_i) = l(S_i) \oplus l(S_{i+\pi})$.

Fluhrer [4] shows that there is actually a second linear function $l'(S) = S[4]_{15} \oplus S[1]_{16} \oplus S[2]_{31} \oplus S[3]_0 \oplus S[4]_{16}$ with the same correlation. We find it interesting that in all the test data sets we have used, admittedly a limited number, our program always "homes in" on $l(S)$ and not $l'(S)$. The existence of this second correlation makes it harder for the program to converge to the correct correlation and explains why more input data is required than would be inferred from previous results such as [5]. We are continuing to explore this area.

The words of the LFSR state are updated according to a bitwise feedback polynomial, but since the wordsize (32 bits) is a power of two, entire words of state also obey the recurrence relation, being related by the 32nd power of the feedback polynomial.

If the two streams Z_i and $Z_{i+\pi}$ were independent, then the correlation probability would be $P(\mathrm{LSB}(Z'_i) = L(S_i)) = 17/32$. However these streams are clearly not independent and, experimentally, we have determined that there is a "second order" effect and in practice the error probability is approximately 0.446, rather than the expected 0.46875. This fortuitous occurrence makes the fast correlation attack more efficient, and counters to some extent the confusion caused by the existence of two correlation functions.

The attack on the LFSR half-cipher proceeds by first gathering approximately 32,000,000 words Z'_i, of which only the least significant bits are utilised in the attack. This requires two segments of a single output stream, separated by π. We then perform fast correlation calculations, to attempt to "correct" the output stream, on different amounts of input varying between 29,000,000 bits and 32,000,000 bits. Empirically, about 2/3rds of these trials will terminate and produce the correct output $L(S_i)$; some of the trials might give an incorrect answer, while others will "bog down", performing a large number of iterations without correcting a significant number of the remaining errors. The sections below describe the fast correlation attack itself in some detail. If the attack is thought to have corrected the output, linear algebra is used to relate this back to the initial state S_0. The sequence $Z'_i = Z_i \oplus Z_{i+\pi}$ can be reconstructed from the initial state to verify that S_0 is correct. If S_0 is incorrect or the attack "bogs down", then a different number of input bits will be tried. Thanks to the numerous optimisations discussed below, a single fast-correlation computation when successful takes about an hour on a 250MHz Sun UltraSPARC (not a particularly fast machine by today's standards) and uses about 70MB of memory. When a computation "bogs down" it is arbitrarily terminated after 1000 rounds, and this takes a few hours. For a particular output set, the full initial state is often

recovered in as little as one hour, and it is very unlikely that the correct state will not be found within a day.

3.3 Increasing the Accuracy of Fast Correlation Attacks

The discussion below applies mostly to LFSRs with low weight feedback, in particular where a trinomial feedback is in use.

A number of papers have been written since [9] applying heuristic techniques to speeding up or increasing the accuracy of the basic technique of fast correlation attacks. These include [3,5,8,11]. We first spent a lot of time examining some of these techniques, and variation in their basic parameters, to gain an intuitive understanding of what is useful and what is not.

The original technique of [9] distinguished between "rounds" and "iterations", where a round started with each of the bits having the same a priori error probability. A new probability was calculated for each bit based on the probabilities of the other bits involved in parity check equations. Subsequent iterations performed the same calculations based on the updated probabilities, until enough bits had error probabilities exceeding some threshhold, or a predetermined number of iterations had been exceeded. We found the arguments in favour of performing iterations unsatisfying, since it seemed that the new probabilities were just self-reinforcing. Eventually, we made structural changes to our program which made it impossible to do iterations, and found an overall increase in accuracy.

The basic correlation algorithm has the error probability P as an input parameter; P is kept constant throughout the computation, and the bit probabilities are reset to P at the beginning of each round. In reality, the error probabilities decrease with each round (at least initially), so this approach results in inaccurate estimates for the bit probabilities. We found that as the real error probability approaches 0.5, then a constant value of P is unlikely to result in a successful attack. The computation is more likely to be successful if P is estimated at each round. For a given P, it is straightforward to calculate the proportion of parity check equations expected to be satisfied by the data. This process is easily reversible, too; having observed the proportion α of parity check equations satisfied, it is easy to calculate the error probability P:[1]

$$\delta \;=\; 1 - 2\alpha, \qquad P \;=\; \tfrac{1}{2}(1 - \delta^{1/3}).$$

Since each round begins by counting parity check equations, it is a simple matter to calculate P for that round. This technique essentially forbids the use of iterations, and obviates techniques like "fast reset", but nevertheless speeds up the attack and increases the likelihood of success.

We felt that having the greatest possible number of parity check equations for each bit was important to the operation of the algorithm, so we performed a one-time brute force calculation to look for low-weight multiples of the feedback

[1] This formula is based on the check equations being trinomials.

polynomial other than the obvious ones (the powers of the basic polynomial). We found a number of them. As well as $x^{127} + x^{64} + 1$, the attack uses

$$x^{16129} + x^{4033} + 1, \qquad x^{12160} + x^{4159} + 1, \qquad x^{12224} + x^{8255} + 1,$$
$$x^{16383} + x^{12288} + 1, \qquad x^{24384} + x^{12351} + 1.$$

and all possible powers of these polynomials. For each bit, the parity checks with that bit at the left, in the middle, and at the right, were all used. For 30,000,000 input bits, an average of 200 parity check equations applied to each bit.

Lastly, we made the observation that relatively early in the computation, a significant number of bits satisfied all of the available parity check equations. We called these *fully satisfied* bits. Experimentally we determined that when more than a few hundred such bits were available, and if the computation was eventually successful, they were almost all correct, so that any subset of 127 of them had a high probability of forming a linearly independent set of equations in the original state bits, which could then be solved in a straightforward manner. Computationally, taking this early opportunity to calculate the answer is a significant performance improvement. In a typical run with 30,000,000 bits of input, 5,040 fully satisfied bits were available after 16 rounds, all of which turned out to be correct, while the full computation required 64 rounds. This is not as great an optimisation as it sounds, because the rounds get faster as the number of bits corrected decreases (see below).

3.4 Increasing the Speed of Fast Correlation Attacks

At the same time as we were analysing the theoretical basis for improvements in the algorithm, we also looked at purely computational optimisations to the algorithm. When the probability of error of individual bits is variable, probability computations are complex and require significant effort for each bit, as well as the requirement to store floating-point numbers for each bit. When the error probability P is assumed the same for all bits at the beginning of a round, the computation is significantly eased. More importantly, the likelihood that a particular bit is in error can be expressed as a threshhold of the number of unsatisfied parity check equations, given the total number of parity check equations for that bit, and the probability P.

The number of parity check equations available for a particular bit is least near the edges of the data set, and increases toward the middle. During the first pass over the data, the number of equations available for each bit is simply counted (this is computationally irrelevant compared to actually checking the equations) and the indexes where this total is different to that for the previous bit is stored. Thus, it requires very little memory to derive the total number of parity checks for a particular bit in subsequent passes. In each round, the first pass over the data calculates (and stores) the number of unsatisfied checks for each bit. From the total proportion of parity checks unsatisfied, P is calculated for this round, and from that, threshhold values above which a bit will be considered to be in error are calculated for each number of parity check equations. When $P <$

0.4 it is approximately correct that more than half of the parity checks unsatisfied implies that the probability of the bit being erroneous is greater than 0.5, and the bit should be corrected. However, when $P > 0.4$, more equations need to be unsatisfied before flipping a bit is theoretically justified. The algorithm's eventual success is known to be very dependent on these early decisions.

A pass is then made through the data, flipping the bits that require it. For each bit that is flipped, the count of unsatisfied parity checks is corrected, not only for that bit, but for each bit involved in a parity check equation with it. The correction factor is accumulated in a separate array so that the correction is applied to all bits atomically. Bits which have no unsatisfied parity checks are noted. In the early rounds, this incremental approach doesn't save very much, but as fewer bits are corrected per round the saving in computation becomes very significant.

Typically another 50% of the overall computation is then saved when the count of fully satisfied bits significantly exceeds the length of the register, and the answer is derived from linear algebra. The net effect of the changes described in this and the previous section is a factor of some hundreds in the time required for data sets of about 100,000 bits over a straightforward implementation. We did not have time to find the speedup for larger data sets, as it would have required too long to run the original algorithm.

4 Attacking the LFG Half-Cipher

This attack derives the initial state $IV = (Y_{16}, \ldots, Y_0)$ of the LFG from outputs of the LFG-half cipher: $V_i = L_i + M_i \bmod 2^{32} = Z_i \oplus N_i$. Much of the analysis is based on dividing the 32-bit words into two 16-bit blocks: $A = A''\|A'$. Note that

$$Y'_{i+17} - Y'_{i+12} - Y'_i \equiv 0 \bmod 2^{16},$$
$$Y''_{i+17} - Y''_{i+12} - Y''_i \equiv f_i \bmod 2^{16},$$

where $f_i \in \{0, 1\}$, denotes the carry bit to the upper half in the sum $(Y_i + Y_{i+12})$.

The value $\mu_i = (Y_{i+17} >> 28)$ chooses M_i from the set $\{Y_{i+1}, \ldots, Y_{i+17}\}$: $M_i = G[1 + (s + \mu_i \bmod 16)]$. The value α_i such that $M_i = Y_{i+\alpha_i}$, is the *multiplexor difference*. We always write μ_i in hexadecimal form, and α_i in decimal form. The particular word chosen depends on μ_i and s, where s is directly related to value of $\mathbf{i} \equiv i \bmod 17$. For example, if $\mu_i = 0$, then $\alpha_i = 12$ unless $\mathbf{i} \in \{4, 5\}$, in which case $\alpha_i = 11$.

4.1 Motivation

The attack exploits a property of outputs (V_i, V_{i+12}) with $\alpha_i = \alpha_{i+12} = 12$. These are called *good pairs*; all other pairs (V_i, V_{i+12}) are *bad pairs*. The initial state can be derived from good pairs using the following observations.

1. If (V_i, V_{i+12}) is good, then $M_i = Y_{i+12}$, and $M_{i+12} = Y_{i+24}$, so

$$\begin{aligned} V'_{i+12} - V''_i &\equiv (Y'_{i+24} + Y''_{i+12}) - (Y''_{i+12} + Y'_i + g_i) \\ &\equiv Y'_{i+24} - Y'_i - g_i \bmod 2^{16}, \end{aligned}$$

   where $g_i \in \{0, 1\}$ is the carry bit from the lower half to the upper half in the sum $V_i = L_i + M_i$. Note that if an attacker is given a good pair, then $(Y'_{i+24} - Y'_i)$ can be derived from $(V'_{i+12} - V''_i)$ by guessing g_i.
2. Every 16-bit half-word Y'_i is a linear function (mod 2^{16}) of the half-word initial state $IV' = (Y'_{16}, \ldots, Y'_0)$. Thus $(Y'_{i+24} - Y'_i)$ is also a linear function (mod 2^{16}) of IV'. We say that the values $(Y'_{i+24} - Y'_i)$ are *linearly independent* (LI) if the linear equations for $(Y'_{i+24} - Y'_i)$ are linearly independent. If the attacker knows a set of 17 LI values $(Y'_{i+24} - Y'_i)$ then the values of IV' can be determined by solving the system of linear equations.
3. Now, having obtained IV', all values Y'_i in the sequence $\{Y'_i\}$ can be computed. For each of the 17 good pairs, the value of Y'_{i+12} allows

$$Y''_i \equiv V'_i - Y'_{i+12} \bmod 2^{16}$$

   to be computed. Computing Y'_i completes the word $Y_i = Y''_i \| Y'_i$. The 17 equations for Y_i (in terms of the complete initial state IV) will also be LI, so this system can be solved to find the initial state, and the attack is complete.

There remain two problems: guessing the 17 carry bits g_i and identifying good pairs.

4.2 Guessing the Carry Bits

The attack will have to try various combinations of values for g_i before the correct carry bits are found. The attack avoids trying all 2^{17} combinations by computing an accurate *prediction* p_i for the value of g_i. Note that if $V'_i < 2^{15}$ then the carry from the sum $(Y''_i + Y'_{i+12} \bmod 2^{16})$ is more likely to be one than zero. That is, we can predict that $g_i = 1$. Conversely, if $V'_i \geq 2^{15}$ then the carry g_i is more likely to be zero than one. Based on this, the attack either sets $p_i = 1$ when $V'_i < 2^{15}$ or sets $p_i = 0$ when $V'_i \geq 2^{15}$. Hence, rather that guessing the carry bits g_i, the attack guesses the 17 *errors* $\epsilon_i = p_i \oplus g_i$. The attack first guesses that there are no errors (all $\epsilon_i = 0$), then one error (one value of $\epsilon_i = 1$), two errors, and so forth. The accuracy of the prediction, $P(p_i = g_i)$, depends on V'_i. Experimental results are shown in Table 1.

Table 1. Experimental approximation to the accuracy of the prediction, $P(p_i = g_i)$, as a function of the four MSBs of V'_i.

The 4 MSBs of V'_i	0,1	2,3	4,5	6,7	8,9	A,B	C,D	E,F
$P(p_i = g_i)$	0.96	0.83	0.7	0.56	0.56	0.69	0.8	0.93

If all the pairs are good then there will be only a small number of errors. When choosing the 17 LI values $(Y'_{i+24} - Y'_i)$, the attack gives preference to values with accurate predictions as there are fewer errors, and the attack will be faster. As shown below, the attack has a small probability of choosing one or more bad pairs. If the correct initial state is not found while the number of errors is small, then this suggests that one of the pairs is bad, so our attack chooses another set of 17 LI values $(Y'_{i+24} - Y'_i)$.

4.3 Identifying Good Pairs

There are 16 possible values for α_i, so we expect good pairs to occur every $16^2 = 256$ words (on average). The problem is identifying good pairs. The trick is to identify *triples* $(V_i, V_{i+12}, V_{i+17})$ with $\alpha_i = \alpha_{i+12} = \alpha_{i+17} = 12$. Bleichenbacher and Meier [1] noted that if $\alpha_i = \alpha_{i+12} = \alpha_{i+17}$, then

$$\Delta_i \stackrel{\text{def}}{\equiv} V_{i+17} - V_{i+12} - V_i \bmod 2^{32} \in \{0, 1, -2^{16}, 1 - 2^{16}\} = \mathcal{A}.$$

A triple of outputs $(V_i, V_{i+12}, V_{i+17})$ that results in $\Delta_i \in \mathcal{A}$ is said to be *valid*, because $\alpha_i = \alpha_{i+12} (= \alpha_{i+17})$ with probability close to one, (which fulfills part of the requirement for a good pair).

Note that $\mu_{i+17} \equiv \mu_{i+12} + \mu_i + c \pmod{16}$, where $c \in \{0, 1\}$, due to the recurrence (1). Hence, the possible combinations for $(\mu_i, \mu_{i+12}, \mu_{i+17})$ that result in $\alpha_i = \alpha_{i+12} = \alpha_{i+17}$ are those given in Table 2 (these are also noted in [1]).

Table 2. The possible combinations for $(\mu_i, \mu_{i+12}, \mu_{i+17})$ that result in $\alpha_i = \alpha_{i+12} = \alpha_{i+17}$

$(\mu_i, \mu_{i+12}, \mu_{i+17})$	(0,0,0)	(0,F,0)	(1,0,1)	(F,0,F)	(F,F,F)
Values of **i**	$\mathbf{i} \notin \{4, 5, 9, 10\}$	$\mathbf{i} = 9$	$\mathbf{i} \in \{9, 10\}$	$\mathbf{i} = 4$	$\mathbf{i} \notin \{4, 9\}$
α_i	12	12	11	12	13

A valid triple that corresponds to a good pair is also said to be good; otherwise the triple is said to be bad. If $\mathbf{i} = 4$, then all valid triples are good, and they are used in the attack. If $\mathbf{i} \in \{5, 10\}$, then all valid triples are bad so these triples are ignored. We currently do not have an efficient method of distinguishing between the cases when $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (0, \mathrm{F}, 0)$ and $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (1, 0, 1)$, so the attack also ignores triples with $\mathbf{i} = 9$.

If $\mathbf{i} \notin \{4, 5, 9, 10\}$, then a valid triple is equally likely to be either good or bad: good when $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (0, 0, 0)$, and bad when $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (\mathrm{F}, \mathrm{F}, \mathrm{F})$. Most of the bad triples are filtered out by examining the values of V'_i and

$$\begin{aligned} \delta_i &\stackrel{\text{def}}{\equiv} V''_{i+17} - V''_{i+12} - V''_i \bmod 2^{16}, \\ \nu_i &\stackrel{\text{def}}{=} ((V''_{i+12} - V'_i \bmod 2^{16}) >> 12) \\ &= \text{the 4 MSBs of } (V''_{i+12} - V'_i \bmod 2^{16}). \end{aligned}$$

Table 3. The probabilities of certain properties being satisfied in the two cases where $(\mu_i, \mu_{i+12}, \mu_{i+17}) \in \{(0,0,0),\ (\mathrm{F},\mathrm{FF})\}$

$(\mu_i, \mu_{i+12}, \mu_{i+17})$	$P(\nu_i \in \{0,1,\mathrm{F}\})$	$P(\delta_i \in \{0,-1\})$	$P(V_i' \geq 2^{15} : \delta_i = 0)$
(0,0,0)	1	0.99	0.85
(F, F, F)	$\frac{3}{16}$	0.51	0.15

The attack discards valid triples with $\mathbf{i} \notin \{4,5,9,10\}$, if

- $\nu_i \notin \{0,1,\mathrm{F}\}$, or
- $\delta_i \notin \{0,1\}$, or
- $V_i' < 2^{15}$ and $\delta_i = 0$.

Following this, $0.99 \times 0.85 = 0.84$ of the good triples remain, while only $\frac{3}{16} \times 0.51 \times 0.15 = 0.024$ of the bad triples remain. Thus, $0.024/0.84 = 0.028$ (one in 36) of the remaining valid triples are bad. The bound on V_i' (when $\delta_i = 0$) can be increased to further reduce the fraction of bad triples to good triples. However, this will also reduce the number of good triples that remain so the attack would require more key-stream.

LFG Half-Cipher Attack Algorithm

1. Find a set of triples with $\mathbf{i} \notin \{5,9,10\}$ and $\Delta_i \in \mathcal{A}$ (valid triples). For $\mathbf{i} \neq 4$, discard triples if
   - $\nu_i \notin \{0,1,\mathrm{F}\}$,
   - $\delta_i \notin \{0,1\}$, or if
   - $\delta_i = 0$ and $V_i' < 2^{15}$.

   For each remaining triple, set $p_i = 1$ if $V_i' < 2^{15}$; else set $p_i = 0$.
2. From these triples, find 17 LI values $(Y'_{i+24} - Y'_i)$, for which $P(p_i = g_i)$ is high.
3. Guess the errors ϵ_i. If the number of errors gets large, then return to Step 2.
4. Compute IV' from $(Y'_{i+24} - Y'_i) \equiv V'_{i+12} - V''_i - (p_i \oplus \epsilon_i) \bmod 2^{16}$.
5. Compute $Y''_i \equiv V'_i - Y'_{i+12} \bmod 2^{16}$, to obtain Y_i.
6. Compute the entire state IV from Y_i. Return to Step 3 if IV produces the incorrect output.

4.4 Complexity

The number of outputs required for the attack is affected by three factors.

1. **The probability that a triple is valid.** Recall that $\mu_{i+17} \equiv \mu_{i+12} + \mu_i + c \,(\mathrm{mod}\ 16)$. To obtain $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (0,0,0)$, it is sufficient to have $\mu_i = 0$, $\mu_{i+12} = 0$ and $c = 0$, so the combination (0,0,0) occurs with probability 2^{-9}. Similarly, $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (\mathrm{F},0,\mathrm{F})$ and $(\mu_i, \mu_{i+12}, \mu_{i+17}) = (\mathrm{F},\mathrm{F},\mathrm{F})$ occur with probability 2^{-9} each.

2. **The probability that a valid triple is good**. Of the good triples with $\mathbf{i} \notin \{4, 5, 9, 10\}$, only 0.84 proceed to Step 2, while all of the good triples with $\mathbf{i} = 4$ proceed to Step 2. So the probability of a good triple getting to Step 2 is $2^{-9} \times (\frac{13}{17} \times 0.84 + \frac{1}{17} \times 1)$.
3. **Finding 17 LI values of $(Y'_{i+24} - Y'_i)$ from the good triples.** Assuming that 17 good triples get to Step 2 there is no guarantee that the values $(Y'_{i+24} - Y'_i)$ are LI. However, we found that a set of 21 values of $(Y'_{i+24} - Y'_i)$ is typically sufficient to find 17 that are LI.

Therefore, the average number of outputs required for the attack on the LFG half-cipher is around

$$21 \cdot \left[\left(\frac{13}{17} \times 0.84 + \frac{1}{17} \times 1 \right) \times 2^{-9} \right]^{-1} \; = \; 15300.$$

There is a large variation in the time/process complexity, as the attacker will have to return to Step 2 if a bad triple has been selected. Our implementation of the attack on a 250MHz Sun UltraSPARC typically takes between 0.1 and 10 seconds.

5 Conclusion

We have demonstrated that attacks on SSC2 are computationally feasible, given a sufficient amount of key-stream. The attack requires portions from a (currently) prohibitive amount of continuous key-stream (around 2^{52} continuous outputs). However, we suggest that the existence of this attack indicates that SSC2 is not sufficiently secure for modern encryption requirements.

References

1. D. Bleichenbacher and W. Meier. Analysis of SSC2. *Fast Software Encryption Workshop, FSE 2001, to be published in the Lecture Notes in Computer Science, Program chair: M. Matsui, Springer-Verlag*, 2001.
2. C. Carroll, A. Chan, and M. Zhang. The software-oriented stream cipher SSC-II. In *Proceedings of Fast Software Encryption Workshop 2000*, pages 39–56, 2000.
3. V. Chepyzhov and B. Smeets. On a fast correlation attack on certain stream ciphers. *Advances in Cryptology, EUROCRYPT'91, Lecture Notes in Computer Science, vol. 547, D. W. Davies ed., Springer-Verlag*, pages 176–185, 1991.
4. S. Fluhrer, P. Crowley, I. Harvey. http://www.cluefactory.org.uk/paul/crypto/ssc2/mail1.txt
5. J. Dj. Golić, M. Salmasizadeh, A. Clark, A. Khodkar, and E. Dawson. Discrete optimisations and fast correlation attacks. *Cryptography: Policy and Algorithms, Lecture Notes in Computer Science, vol. 1029, E. Dawson, J. Golić eds., Springer*, pages 186–200, 1996.
6. P. Hawkes and G. Rose. Correlation cryptanalysis of SSC2, 2000. Presented at the Rump Session of CRYPTO 2000.

7. P. Hawkes and G. Rose. Exploiting multiples of the connection polynomial in word-oriented stream ciphers. *Advances in Cryptology, ASIACRYPT2000, Lecture Notes in Computer Science, vol. 1976, T. Okamoto ed., Springer-Verlag*, pages 302–316, 2000.
8. T. Johansson and F Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. *Advances in Cryptology, EUROCRYPT'99, Lecture Notes in Computer Science, vol. 1592, J. Stern ed., Springer-Verlag*, pages 347–362, 1999.
9. W. Meier and O. Staffelbach. Fast correlation attacks on certain stream ciphers. *Advances in Cryptology, EUROCRYPT'88, Lecture Notes in Computer Science, vol. 330, C. G. Günther ed., Springer-Verlag*, pages 301–314, 1988.
10. W. Meier and O. Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3):159–176, 1989.
11. M. Mihaljević and J Golić. A comparison of cryptanalytic principles based on iterative error-correction. *Advances in Cryptology, EUROCRYPT'91, Lecture Notes in Computer Science, vol. 547, D. W. Davies ed., Springer-Verlag*, pages 527–531, 1991.