# Improved Cryptanalysis of the Common Scrambling Algorithm Stream Cipher

Leonie Simpson[1], Matt Henricksen[2], and Wun-She Yap[2]

[1] Information Security Institute,
Queensland University of Technology,
GPO Box 2434, Brisbane Qld 4001, Australia
lr.simpson@qut.edu.au
[2] Institute for Infocomm Research,
A*STAR, Singapore
{mhenricksen,wsyap}@i2r.a-star.edu.sg

**Abstract.** This paper provides a fresh analysis of the widely-used Common Scrambling Algorithm stream cipher (CSA-SC). Firstly, a new representation of CSA-SC with a state size of only 89 bits is given, a significant reduction from the 103 bit state of a previous CSA-SC representation. Analysis of this 89-bit representation demonstrates that the basis of a previous guess-and-determine attack is flawed. Correcting this flaw increases the complexity of that attack so that it is worse than exhaustive key search. Although that attack is not feasible, the reduced state size of our representation makes it obvious that CSA-SC is vulnerable to several generic attacks, for which feasible parameters are given.

**Key words:** Digital Video Broadcasting, Common Scrambling Algorithm, Stream Cipher, Cryptanalysis

## 1 Introduction

The Digital Video Broadcasting Common Scrambling Algorithm (CSA) has been used to encrypt European cable digital television signals since 1994. It was specified by the European Telecommunication Standards Institute (ETSI), and the proprietary algorithm was distributed to cable TV subscribers in the form of a hardware chip. Although some high-level details appear in patents [3], the algorithm has never officially been revealed. In 2002, a software program that implemented the algorithm was released in binary form. This was reverse engineered by hackers who released the details of the CSA algorithm.

The CSA algorithm can be considered as the application of two cipher layers: a stream cipher layer and a block cipher layer. For encryption, the block cipher is applied first, followed by the stream cipher layer. For decryption, the stream cipher layer is applied first, followed by the block cipher layer. Both the block and stream ciphers are initialized using the same 64-bit key. We do not consider the block cipher component within this paper. The stream cipher component

is a binary additive stream cipher. We refer to the keystream generator for the stream cipher component of the CSA algorithm as CSA-SC.

The CSA-SC structure comprises two nonlinear Feedback Shift Registers (FSRs), a combiner with memory and an output function. In the patent application [3], the total internal state size of CSA-SC is described as 107 bits. In previous analysis of CSA-SC, Weinmann and Wirt [8] showed that it can be modelled using 103 bits, and note that the period of the keystream produced by CSA-SC is upper bounded by $2^{103}$. In this paper, we provide a new representation of CSA-SC that uses only 89 state bits. Consequently the maximum CSA-SC keystream period must be much less than the $2^{103}$ bits asserted by Weinmann and Wirt [8], with an upper bound of $2^{89}$ bits. This significant reduction in state size also has implications for the security of CSA-SC.

Weinmann and Wirt [8] presented an analysis of CSA-SC and proposed a guess-and-determine attack with complexity less than $2^{45}$, based on their 103 bit CSA-SC representation and predicated on the state cycle structure of one of the FSRs during keystream generation. They claimed that the state cycle structure for this FSR consists of many leading paths and short cycles, with experimental simulation to support this conjecture. However, in examining the cycle structure of the FSR when developing our 89-bit model, we identified that there are no leading paths to cycles, and short cycles were not readily located, implying that the Weinmann-Wirt attack can not work as claimed.

We present our model of the CSA-SC in Section 2. In Section 3, we provide theoretical observations about the state update functions used in CSA-SC. These observations contradict the results presented by Weinmann and Wirt [8], but show that there are security vulnerabilities that can be exploited in cryptanalytic attacks. Section 4 describes an exploration of the FSR state cycle structure. This is motivated by the discrepancy between our observations and the results presented in [8]. Section 5 discusses several possible attacks on CSA-SC. In Section 5.1, the analysis of CSA-SC in [8] is summarized, and the problem with their attack is discussed. In Section 5.2, the vulnerability of CSA-SC to time-memory tradeoff attacks is demonstrated. Section 6 presents some closing remarks on the security of CSA-SC.

## 2 Specification of the CSA-SC

CSA-SC comprises two FSRs and a combiner with memory. These are denoted FSR-A, FSR-B and FSM-C, respectively. For our representation of CSA-SC, FSR-A and FSR-B each have ten stages, with each stage containing one four-bit word (a nibble). The combiner FSM-C consists of two stages, each containing a nibble, and a single-bit carry. The total state size is 89 bits. Figure 1 shows a high-level view of the relationships between components of the CSA-SC during keystream generation.

During keystream generation, FSR-A is autonomous. The state update function for FSR-A is nonlinear, with the output of the s-box $S_A$ used in calculating the next state value. Nonlinear outputs from FSR-A are also used as input to

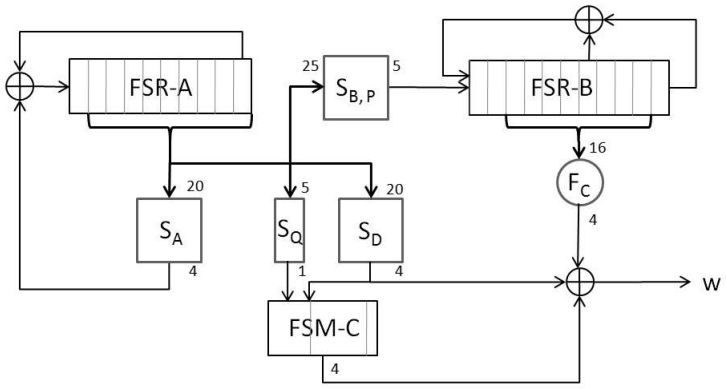


**Fig. 1.** High-level view of the relationship between components during CSA-SC keystream generation

the update functions of both FSR-B and FSM-C, by providing all of the input to s-boxes $S_B$, $S_D$, $S_P$, and $S_Q$. The outputs of $S_B$ and $S_P$ are used to modify FSR-B, and the outputs of $S_D$ and $S_Q$ are used to modify FSM-C. The specific state update functions for each component and the keystream output function are described in greater detail in Section 2.1.

The notation used in this paper is as follows. Let $A^t$ represent the contents of FSR-A at time $t$. Then $A^t_{i,j}$ represents bit $j$ of the $i$th stage of FSR-A at time $t$, where $i \in \{0, 1, \ldots, 9\}$ and $j \in \{0, 1, 2, 3\}$. Similarly, $B^t_{i,j}$ represents bit $j$ of the $i$th stage of FSR-B at time $t$. At time $t$, the contents of the two four-bit stages of FSM-C are denoted $D^t_{i,j}$, where $i \in \{0, 1\}$ and $j \in \{0, 1, 2, 3\}$, and the contents of the one-bit carry are denoted $c^t$. According to ETSI conventions, the most significant bit of a stage is denoted by index 0. Binary addition and modular addition in $\mathbb{Z}_{2^4}$ are represented by $\oplus$ and $\boxplus$ operators, respectively. $ROL_x$ represents word rotation to the left by $x$ bits. $c||D$ represents the concatenation of bit $c$ and word D. For example, $0||1001 = 01001$.

### 2.1 Generating keystream

When the keystream generator is clocked at time $t$, FSR-A, FSR-B and FSM-C are simultaneously clocked. FSR-A is autonomous, and contributes to the state update functions of FSR-B and FSM-C. Nonlinear combinations of values stored in FSR-A stages, obtained through the use of various s-boxes, are used in the state update functions of all components. S-boxes $S_A$, $S_B$ and $S_D$ each take twenty-bit inputs from FSR-A and provide 4-bit outputs. S-boxes $S_P$ and $S_Q$ take 5 bits of input from FSR-A, and each produce a 1-bit output. Specific details

regarding the s-boxes are contained in Appendix A. The state update functions for the CSA-SC components are as follows:

$$\begin{aligned} A_i^t &= A_{i-1}^{t-1} && 1 \le i \le 9 \\ A_0^t &= A_9^{t-1} \oplus S_A(A^{t-1}) \end{aligned}$$

$$\begin{aligned} B_i^t &= B_{i-1}^{t-1} && 1 \le i \le 9 \\ B_0^t &= ROL_{S_P(A^{t-1})}(B_6^{t-1} \oplus B_9^{t-1} \oplus S_B(A^{t-1})) \end{aligned}$$

$$\begin{aligned} D_1^t &= D_0^{t-1} \\ c^t||D_0^t &= \begin{cases} c^{t-1}||D_1^{t-1} & \text{if } S_Q(A^{t-1}) = 0 \\ S_D(A^{t-1}) \boxplus D_1^{t-1} \boxplus c^{t-1} & \text{if } S_Q(A^{t-1}) = 1 \end{cases} \end{aligned}$$

The keystream is produced as a series of 2-bit words. The contents of FSR-A, FSR-B and FSM-C all contribute to the keystream output word $z^t$. To facilitate effective cryptanalysis, we consider the formation of $z^t$ based on an intermediate word, denoted $w^t$. The 4-bit intermediate word $w^t$ and the 2-bit keystream output $z^t$ are obtained as follows:

$$\begin{aligned} w^t &= S_D(A^{t-1}) \oplus F_C(B^{t-1}) \oplus D_1^{t-1} \\ z^t &= f(w^t \gg 2)||f(w^t \bmod 2^2) \end{aligned}$$

where

$$f(x) = \begin{cases} 0 \text{ if } x = 0 \text{ or } x = 3 \\ 1 \text{ if } x = 1 \text{ or } x = 2 \end{cases}$$

The function $F_C$ produces four bits of output, with each output bit formed from a linear combination of four bits from FSR-B. Specifically, $F_C(B^t) = (B_{2,0}^t \oplus B_{5,1}^t \oplus B_{6,2}^t \oplus B_{8,3}^t)||(B_{5,0}^t \oplus B_{7,1}^t \oplus B_{2,3}^t \oplus B_{3,2}^t)||(B_{4,3}^t \oplus B_{7,2}^t \oplus B_{3,0}^t \oplus B_{4,1}^t)||(B_{8,2}^t \oplus B_{5,3}^t \oplus B_{2,1}^t \oplus B_{7,0}^t)$.

### 2.2 A Note on Previous Representations

In the DVB patent [3], the CSA-SC algorithm is defined as having a state of 107 bits. This representation facilitates an efficient hardware implementation. The CSA-SC representation of Weinmann and Wirt [8] reduced the state size from 107 to 103 bits.

Our representation obtains a further reduction in the state size to 89 bits, by removing the 4-bit memories $X$, $Y$, and $Z$ and one-bit memories $p$ and $q$ from the representation used by Weinmann and Wirt. These memories hold the outputs of s-boxes at time $t$, and are used in the state update function at time $t+1$ to form feedback. In this representation, none of the s-boxes use the final stage of FSR-A, $A_9$. When the state update function is applied to FSR-A, the contents of $A_{0..8}^t$ are shifted to become $A_{1..9}^{t+1}$. All of the values required to calculate the feedback at time $t$ remain in the FSR at time $t+1$, so the memories, while useful in constructing efficient hardware, are not required in an equivalent representation

of the shift register. In the equivalent representation, the indices of the stages used as inputs to the s-boxes must be incremented by one.

Note that in the original representation, the initial value of all memories is zero. For our representation, this necessitates a special case for the first clock of the key initialization process where the output of the s-boxes must be treated as zero irrespective of their inputs. Although this may not be the most efficient hardware implementation, it permits a cryptographically equivalent representation of CSA-SC using only 89 bits.

Neither our work nor that of Weinmann and Wirt considers the key initialization during cryptanalysis, so we omit the details of the initialisation process in this paper. There are several errors in the specification given in the work of Weinmann and Wirt [8], which we correct in Appendix A of this paper.

## 3 Some observations on CSA-SC

In this section, we make some observations regarding CSA-SC. Firstly, in Section 3.1, we show that during keystream generation the state update functions of both FSR-A and FSR-B are invertible. That is, given the state of the two FSRs at time $t+1$, the corresponding FSR states at time $t$ can be uniquely determined. This contradicts the claims of leading paths to short cycles made by Weinmann and Wirt [8], motivating our exploration of the FSR-A state cycles presented in Section 4. Secondly, in Section 3.2 we make observations regarding the ratio of the CSA-SC state size to the key size, which indicates a vulnerability to a generic style of attack.

### 3.1 State update functions during keystream generation

The state update functions for FSR-A and FSR-B are invertible. As FSR-A is autonomous during keystream generation, but FSR-B is dependant on FSR-A, it is necessary to establish that the state update function for FSR-A is invertible before examining the state update function for FSR-B. Following this, the conditions under which the FSM-C may be inverted are also presented.

The state update function for FSR-A makes use of $S_A$, a $20 \times 4$ s-box. Although $S_A$ itself is not bijective, the FSR-A state update function is nevertheless invertible because (rearranging the state update function in Section 2.1):

$$\begin{aligned} A_i^{t-1} &= A_{i+1}^t && 0 \leq i \leq 8 \\ A_9^{t-1} &= A_0^t \oplus S_A(A^{t-1}) \end{aligned}$$

That is, for the inversion, the register contents are shifted back one stage, rather than forward, and the contents of the last stage, $A_9$, are computed from the contents of $A_0^t$ and the output of $S_A$ at time $t-1$. Now given that $S_A(A^{t-1})$ takes as input 20 bits from $A^{t-1}$, including the two bits $A_{9,0}^{t-1}$ and $A_{9,1}^{t-1}$, this initially appears to cause a circular dependancy. However, if $S_A$ is considered as the concatenation of four 5-input Boolean functions, then the ouput of each

of these functions can be computed individually and the dependency avoided. Table 3 in Appendix A shows the stages of FSR-A which provide the inputs to each of the four Boolean functions. $A_{9,0}^{t-1}$ and $A_{9,1}^{t-1}$ depend upon the input sets $s_3$ and $s_2$, respectively. Only bits in stages 1-6 and 8 of $A^{t-1}$ (equivalent to stages 2-7 and 9 of $A^t$), are required, and these are already known. $A_{9,2}^{t-1}$ and $A_{9,3}^{t-1}$ depend upon the input sets $s_1$ and $s_0$, respectively. These input sets include $A_{9,0}^{t-1}$ and $A_{9,1}^{t-1}$. Therefore, if $A_{9,0}^{t-1}$ and $A_{9,1}^{t-1}$ have been calculated, then $A_{9,2}^{t-1}$ and $A_{9,3}^{t-1}$ can also be calculated, obtaining the state $A^{t-1}$ in its entirety. As the state update function for FSR-A is invertible, if the state of FSR-A is known at time $t$, then all future and previous states of FSR-A during keystream generation can be easily calculated.

It follows from the invertibility of FSR-A and the use of only a linear combination of the stages of FSR-B in the state update function for FSR-B that the state update function of FSR-B can also be inverted, once $A^{t-1}$ is obtained, as follows:

$$\begin{aligned} B_i^{t-1} &= B_{i+1}^t & 0 \le i \le 8 \\ B_9^{t-1} &= ROL_{3-S_P(A^{t-1})}(B_0^t) \oplus B_6^{t-1} \oplus S_B(A^{t-1}) \\ &= ROL_{3-S_P(A^{t-1})}(B_0^t) \oplus B_7^t \oplus S_B(A^{t-1}) \end{aligned}$$

That is, given the internal state of FSR-A and FSR-B at time $t$ during keystream generation, all previous and future state values for these two feedback shift registers can be calculated.

The final component to consider is FSM-C. From the FSM-C state update function, $D_0^{t-1} = D_1^t$. However, the value of $c^{t-1}||D_1^{t-1}$ is dependent on the values of $c^t$, $D_0^t$ and $A^{t-1}$. When $S_Q(A^{t-1}) = 0$, then given $D^t$ and $c^t$, it is obvious that $c^{t-1}||D_1^{t-1} = c^t||D_0^t$. However, when $S_Q(A^{t-1}) = 1$, the calculation of $c^{t-1}||D_1^{t-1}$ is more complex, due to the use of integer addition rather than XOR.

Consider the case where $S_Q(A^{t-1}) = 1$. As $c^t||D_0^t = S_D(A^{t-1}) \boxplus D_1^{t-1} \boxplus c^{t-1}$, clearly $D_1^{t-1} \boxplus c^{t-1} = c^t||D_0^t - S_D(A^{t-1})$, where the addition and subtraction are integer rather than bitwise operations. There are two possible values for $c^{t-1}$, so this provides two possible values for $c^{t-1}||D_1^{t-1}$.

The intermediate keystream word $w^t$ is given by $w^t = S_D(A^{t-1}) \oplus F_C(B^{t-1}) \oplus D_1^{t-1}$, and $w^t$, $S_D(A^{t-1})$ and $F_C(B^{t-1})$ are all known, but $D_1^{t-1}$ is unknown. However, the sum $D_1^{t-1} \boxplus c^{t-1}$ is known. The contribution of $c^{t-1}$ to $D_1^{t-1} \boxplus c^{t-1}$ is in the least significant bit, but as the addition is integer addition, this raises the possibility of carry to the next bit position, and so on. That is, if $c^{t-1} = 1$ and the least significant bit of $D_1^{t-1} = 1$, then the least significant bit of the integer sum will be 0, and the influence of $c^{t-1}$ is carried to the next position. However, where the least significant bit of the integer sum is 1 (that is, the sum is an odd value), clearly the value of $c^{t-1}$ and the value of the least significant bit of $D_1^{t-1}$ are not the same, so there is no possibility that the influence of $c^{t-1}$ extends beyond that least significant bit position. The 2-bit keystream output $z^t$ is useful in discovering whether the two least significant bits in $D_1^{t-1}$ are the

same (00 or 11) or different (01 or 10). Combining this knowledge with a known odd value of $D_1^{t-1} \boxplus c^{t-1}$ enables a unique value of $c^{t-1}||D_1^{t-1}$ to be determined.

Note that FSM-C is the only section of the CSA-SC internal state where the state cycle mapping may involve branching. If $S_Q(A^{t-1}) = 0$ there is a unique previous state. However, if $S_Q(A^{t-1}) = 1$ then a unique previous state exists only if $D_1^{t-1} \boxplus c^{t-1} = c^t||D_0^t - S_D(A^{t-1})$ is odd, and known keystream bits can be used to determine this unique state. Otherwise, there are two possible prior states for the combiner.

### 3.2 The ratio of state size to key size

The CSA-SC key initialization uses the combination of a 64-bit key and 64-bit IV to populate the 89-bit state in preparation for keystream generation. That is, the initialisation function takes 128 bits of input and produces an 89-bit output: the CSA-SC initial state at the start of keystream generation. Although this paper does not consider the specific details of the initialisation function, clearly there exist multiple key-IV pairs that produce the same internal state at the start of keystream generation, and hence the same keystream. That is, the use of different keys does not guarantee the production of different keystreams. Even for a single 64-bit key, clearly there are multiple IVs for which the initial 40-bit state of FSR-A will be the same. As the nonlinearity of the functions used in CSA-SC is largely determined by the contents of FSR-A, CSA-SC may be vulnerable to divide and conquer attacks which target FSR-A.

The small CSA-SC state size also indicates a potential vulnerability to time-memory-data (TMD) tradeoff attacks. These known-plaintext attacks can be used to identify either the internal state of CSA-SC, or the key. These attacks are discussed in greater detail in Section 5.

## 4 Exploring the state cycles of FSR-A

In Section 3.1, the state update function of FSR-A is shown to be invertible. Therefore every state of FSR-A has exactly one previous state and exactly one successor state. Consequently, there is either one cycle of length $2^{40}$ in the FSR-A state space or there are multiple disjoint cycles. It is possible that some of these may be short cycles. There can be no overlapping cycles, and there are no cycles with leading paths.

Floyd's algorithm [6] can be used to detect cycles. This simple algorithm, when applied to FSR-A of CSA-SC, detects a single cycle using the following steps:

**Algorithm FA**

1. Initialize two instances $A_0$ and $A_1$ of FSR-A with the same initial state $s$
2. Set counter $l$ to 0.
3. Do
   (a) Clock $A_0$ once. Increment counter $l$.

(b) Clock $A_1$ twice.
while state($A_0$) is not equal to state($A_1$).
4. Output $l$ as the length of the cycle.

Applying Floyd's algorithm to comprehensively map all of the cycles produced by FSR-A using the following algorithm raises a problem in detemining which values of $s$ to select.

**Algorithm A**

1. Set $t$ to $2^{40}$.
2. Do
   (a) Choose unique value of $s$.
   (b) Invoke Floyd's algorithm (Algorithm FA) for $s$, which outputs cycle length $l$ for state $s$.
   (c) Decrement $t$ by the value of $l$.
   while $t > 0$.

A naive memoryless approach is to begin with $s = 0$ and increment $s$ until $s = 2^{40} - 1$. This which ensures that all cycles are mapped, but the running time of the process, at $O(2^{80})$, makes this infeasible.

A modification to the algorithm, using a time-memory tradeoff, tracks the state values visited (using a one-bit flag per state). During each invocation of Floyd's algorithm, the value $s$ is chosen from the complement of the set of visited states. The running time of this algorithm is $2^{40} \cdot c$, but the storage requirement is $2^{40} \times \log_2 2^{40}$ bits = 2.3 terabytes. In practice, this version of the algorithm must be implemented using hard disks, which have high latency, so that the constant $c$ becomes quite large. The storage requirements can be improved by tracking and storing only "distinguished points". For example, states with an 8-bit prefix consisting of 0 bits may be considered "distinguished". This reduces disk usage by a factor of $2^8$. However, the algorithm will not detect any cycles that traverse only non-distinguished points. Since Weinmann and Wirt [8] claim the existence of small cycles, we do not want to take this approach.

A possible compromise is to use the first approach, in which at iteration $i$, $s_i = i$, but with a slight modification to include early stopping criteria. If Floyd's algorithm traverses over state $j < i$ at iteration $i$, then the cycle has been visited previously and the algorithm aborts early. Similarly, if the cycle length $l$ is larger than the state space not so far searched, then the algorithm has rediscovered a cycle and aborts.

Given that we know there are no leading paths, the algorithm can be optimized by using a single instance of FSR-A, with a stopping condition that a cycle has been found when the starting point is traversed for the second time.

**Algorithm FA-M**

1. Initialize an instance $A$ of FSR-A with the initial state $s$.
2. Set counter $l_c$ to 0, $l_{max}$ to $t$.
3. Do
   (a) Clock $A$ once.

(b) Increment counter $l_c$.
(c) If $l_c > l_{max}$ then abort.
while state($A$) is not equal to initial state $s$.
4. Output $l_c$ as the length of the cycle

The running time of the FA-M algorithm varies depending on the length of the cycles that it finds. The results of our search are shown in Table 1. They were generated using several Intel Core Duo machines. Each core is capable of iterating through $2^{38}$ states per day.

**Table 1.** Cycles identified in FSR-A State Space

| Cycle number | Cycle length | Cycle length (log 2) |
|---|---|---|
| 1 | 307,080,986 | 28.19 |
| 2 | 783,472,466 | 29.52 |
| 3 | 10,152,192,874 | 33.24 |
| 4 | 14,199,085,442 | 33.72 |
| 5 | 36,257,653,742 | 35.08 |
| 6 | 78,922,935,703 | 36.20 |
| 7 | 225,691,600,544 | 37.72 |
| 8 | 308,063,543,688 | 38.16 |
| 9 | 424,911,536,585 | 38.63 |
| Total | 1,099,289,102,030 | 39.9997 |

The identification of nine large cycles in conjunction with the observation that the update function is invertible provides strongly contradictory evidence to the claims of Weinmann and Wirt [8] that 98% of the state space of FSR-A can be partitioned into very short cycles.

## 5 Cryptanalysis of CSA-SC

Observations made in Section 3.2 indicate CSA-SC may be vulnerable to two common styles of attack. The dependence of other components on the autonomous 40-bit FSR-A for nonlinearity indicates the potential for divide and conquer style attacks which target FSR-A. The ratio of the state size to the key size indicates a vulnerability to a generic style of attack known as Time-Memory Tradeoff (TMTO) attack. The attack in by Weinmann and Wirt [8] targets FSR-A, and is reviewed in Section 5.1. The application of TMTO attacks to CSA-SC is discussed in Section 5.2.

### 5.1 The Weinmann and Wirt Attack

A guess-and-determine attack on CSA-SC which targets FSR-A is presented by Weinmann and Wirt [8]. The aim of the attack is to recover the internal state of

the cipher during keystream generation, when FSR-A is autonomous. The attack complexity is claimed to be less than $2^{45}$. We explain the flaw in this attack and show that, when the flaw is corrected, the attack performance is actually worse than exhaustive key search.

The attack is performed in three phases. In the first phase, the attacker guesses 53 bits of state comprising FSR-A, FSM-C and the 4-bit memory X used by Weinmann and Wirt [8] for their 103-bit CSA-SC representation. Because each output bit of $F_C$ is a linear combination of bits within FSR-B, and these output bits are linearly combined to form the keystream, a system of equations can be formed relating the keystream bits to the unknown contents of FSR-B. The second phase of the attack solves this system of equations using Gaussian elimination. The third phase of the attack comprises consistency checking to establish the veracity of guesses made in the first phase. If the consistency checking fails, the guess in the first phase is considered incorrect and a new guess is made. Otherwise, the attack terminates and the combination of the 53-bit guess and the solution to the equation system is used to recover the initial internal state.

The cost of the first phase is $2^{53}$ operations. In the second phase, a system of equations containing 60 equations in 40 unknowns is developed, which can be solved using Strassen's algorithin in $2^{17.7}$ operations. The cost of the third phase is negligible. The total complexity of the state recovery attack is therefore around $2^{70.7}$ operations, which is about one hundred times worse than a brute-force key search on the 64-bit key.

Weinmann and Wirt [8] claimed to have identified numerous short cycles produced by the FSR-A feedback function. They performed 10,000 random initializations of the cipher, and found that for 98.4% of cases, those key-IV pairs led to FSR-A state cycles with lengths of between 108 and 121,992. This lead to the assumption that for any key-IV pair, the effective state space for FSR-A is equal to the sum of the lengths of those short cycles, with 98.4% probability. An attacker does not know the key, so must guess FSR-A states from all of the points on all of the short cycles. Ignoring leading paths, this gives a total of 313,169 possibilities. Therefore, Weinmann and Wirt[8] claim that the cost of the first phase is reduced to $2^{19} \times 2^9$, where the second term is the cost of guessing the memories and registers in FSM-C.

The optimisation of the guessing phase of the divide and conquer attack is necessary in order for the attack to be faster than exhaustive search. However, both the theoretical observation in Section 3.1 that no leading paths exist because the FSR-A feedback function is invertible, and the empirical results in Section 4 that demonstrate the FSR-A state cycles form a small number of disjoint large cycles show that the basis for the optimisation is unfounded. Thus the performance of Weinmann and Wirt's attack is worse than exhaustive key search, unless further optimizations are identified.

### 5.2 Time-Memory tradeoff attacks

As the TMTO approach is well known, it is not described here. Instead, we refer the reader to the work of Hong and Sarkar [7] for a description of the phases of TMTO attacks. Our analysis aims to determine the feasibility of applying TMTO attacks to CSA-SC given the constraints on the amount of keystream available to an attacker. The specification for Digital Video Broadcasting indicates that keystream is propagated at a maximum rate of 64 Mb/s, and that rekeying occurs at least every 120 seconds. Therefore, for a single key-IV pair, assume that an attacker has access to about $D = 2^{33}$ bits of data. We consider possible tradeoffs for two styles of TMTO.

The first style of TMTO attack is one in which the attacker attempts to invert the CSA-SC keystream to recover the internal state. For this style of attack, the attacker must satisfy the time-memory-data tradeoff curve $N^2 = TM^2D^2$ for $T \geq D^2$ and $T < 2^K$, where $N = 2^{89}$ is the total number of states, $T$ is the time taken to execute the attack, and $M$ is the amount of memory required to store precomputed tables. The value $2^K = 2^{64}$ represents the computational effort required to launch a brute force attack. The attacker is unable to make use of all available data since $T \geq D^2$ implies that $T > 2^K$. Reasonable parameters are therefore $D = 2^{25}, T = 2^{50}$ and $M = 2^{39}$, for which the attacker requires around 6 terabytes of disk space. This is feasible by today's standards.

If the key initialization function was invertible, then the recovered internal state could be used to derive the key. However, this does not seem to be the case. Therefore this attack is of limited use, since frequent key-IV rekeying is mandatory within the DVB specification, and this attack must be performed on the keystream segment obtained after each rekeying.

The second style of TMTO attack attempts to invert the CSA-SC keystream to recover the key. For this style of attack, the state size is immaterial. The same time-memory-data tradeoff curve holds, but $N$ now refers to the number of possible key+IV values, rather than the number of possible internal states. Here, $N = 2^{64+64}$. Taking the Dunklemann-Keller approach [4], the attacker prepares different tables for many IVs in the precomputation phase. This removes the restriction that $T \geq D^2$, at the expense of reducing the success rate of the attack if the right IVs are not used. Due to the larger size of $N$, the computational complexity of this attack is inferior to the first, with one possible parameter set being $D = 2^{48.5}, T = 2^{53}, M = 2^{53}$. However, as this is key recovery rather than state recovery, the stipulation for frequent rekeying does not present a limitation. Therefore, any proposed key recovery attack on CSA-SC with time complexity greater than T=$2^{53}$ should be regarded as unnecessary unless the data and memory requirements are much less than those given for this TMD attack.

## 6 Discussion and Conclusion

In this paper we provide a new representation of CSA-SC that uses only 89 state bits, a significant reduction over the 107 bits and 103 bits used for previous

representations. Theoretical observations about the state update functions of components of CSA-SC contradict the empirical results presented in previous research [8], motivating an exploration of the state cycles for FSR-A.

Our FSR-A state-cycle findings raise doubts about the validity of the optimisation required in order for the divide and conquer attack presented by Weinmann and Wirt [8] to be successful. It appears that the complexity of that state recovery attack is not around $2^{45}$, as claimed, but in fact worse than exhaustive key search. Even applying their approach to our representation of CSA-SC, where the state size is reduced because the memory $X$ is redundant, results in an overall attack complexity of about $2^{66.7}$ operations. This is about thirteen times worse than brute force attack and several orders of magnitude worse than TMTO attacks, although with the memory requirement is less than for the TMTO attacks.

The reduction in the state space obtained in our model indicates that CSA-SC is vulnerable to TMTO attacks. Given a keystream segment produced from a single key-IV pair, a state recovery attack is possible with data, time and memory parameters of $D = 2^{25}, T = 2^{50}$ and $M = 2^{39}$, respectively. Additionally, for an increased data value obtained by taking keystream segments formed from a single key, but possibly multiple known IVs, a key recovery attack is possible, with data, time and memory requirements of $D = 2^{48.5}, T = 2^{53}, M = 2^{53}$, respectively. This application of a generic attack style to CSA-SC shows that CSA-SC is vulnerable to cryptanalytic attack.

The attacks discussed in this paper made no use of the CSA-SC initialisation process. Exploring the initialisation process may reveal weaknesses that will lead to improved attacks on this cipher. Additionally, it may be possible to improve the performance of divide and conquer attacks targeting FSR-A by reducing the complexity of determining whether guessed FSR-A and FSM-C states are correct. This could be accomplished, by using a distinguisher, and only proceeding to solving the system of equations to recover the contents of FSR-B for a correct guess.

We examined the cipher with respect to differential and linear attacks, and to guess-and-determine attacks. Even though the s-boxes are far from optimal with respect to differential and linear attacks, the fact that in each clock cycle, half of the bits in FSR-A are passed through s-boxes makes it difficult to utilize the s-box biases; ie. the diffusion in the register is good. Likewise, this means that a large number of bits must be guessed in order to determine a single nibble in the register, and a straightforward guess-and-determine approach is ineffective in reducing the complexity of an attack below $2^{53}$ operations, even considering the effective reduced state size.

Although there are generic attacks that apply to CSA-SC, it appears that the attack strategy of Weinmann and Wirt [8] does not succeed in key recovery with better complexity than brute force.

**Acknowledgements** Thanks to anonymous referees for their valuable comments. Thanks also to Yian Chee Hoo for his spare computer cycles.

## References


1. Anonymous. *CSA - known facts and speculations*, http://csa.irde.to, 2003.
2. Daniel J. Bernstein. *Costs of cryptanalytic hardware*, Posting to ECRYPT eSTREAM forum, August 21, 2005. Available at : http://www.ecrypt.eu.org/stream/phorum/read.php?1,95,95#msg-95
3. Simon Bewick. *Descrambling DVB data according to ETSI common scrambling specificiation.* UK Patent Application GB2322995A, 1998.
4. Orr Dunkelman and Nathan Keller. *Treatment of the Initial Value in TMTO Attacks.* SASC 2008: The State of the Art of Stream Ciphers, Lausanne, Switzerland, 2008. pages 249-258.
5. FFDeCSA 1.0.0 implementation.
   *Available at http://www.dvbsupport.net/download/index.php? act=view&id=129*
6. Robert Floyd. *Non-deterministic Algorithms*, Journal of the Association for Computing, Volume 4, Number 14, March 1967, pages 636-644.
7. Jin Hong and Palash Sarkar. *New Applications of Time Memory Data Tradeoffs*, Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Lecture Notes in Computer Science vol 3788, Springer, 2005, pages 353–372.
8. Ralf-Phillip Weinmann and Kai Wirt. *Analysis of the DVB Common Scrambling Algorithm.* Communications and Multimedia Security, Proceedings of the 8th IFIP TC-6 TC-11 Conference on Communications and Multimedia Security (CMS 2004), Springer-Verlag.


## A CSA-SC S-boxes

CSA-SC has not been published by ETSI. Information has been revealed in patents [3] and by reverse engineering of implementations; see, for example, [5]. The specification by Weinmann and Wirt [8] is the best academic description of CSA-SC in the public literature to date but contains multiple errors; including a misprint in the table that specifies inputs from FSR-A into the s-boxes, and incorrect ANFs for the component boolean functions of the s-boxes. We give the (hopefully) correct versions below.

Table 2 describes the s-boxes referred to in Section 2. Each s-box is built from 5-input boolean functions. S-boxes $S_A$, $S_B$ and $S_D$ are each constructed from four boolean functions, while s-boxes $S_P$ and $S_Q$ are each built from a single boolean function. These boolean functions are denoted $F_i(s_j)$ in the following tables, where $i$ denotes the function index and $s_j$ denotes the $j$th set of FSR-A positions which provide the five inputs to $F_i$. The FSR-A positions for the 5 inputs $x_i, 0 \leq i \leq 4$ are given in Table 3. The boolean functions are given in Algebraic Normal Form in Table 4. For example, the most significant output bit of s-box $S_A$ is $F_6(s_3)$, where $F_6$ is $1+x_0+x_1+x_5+x_0 \cdot x_4 ... + x_0x_1x_2x_3x_4x_5$, and from the definition of $s_3$ in Table 3, $x_0 = A_{3,3}, x_1 = A_{1,1}, x_2 = A_{2,3}, x_3 = A_{4,2}$ and $x_4 = A_{8,0}$.

**Table 2.** S-boxes used in our model of CSA-SC

| S-box | Input Size (bits) | Output Size (bits) | Output |
|---|---|---|---|
| $S_A$ | 20 | 4 | $F_6(s_3)\|\|F_4(s_2)\|\|F_3(s_1)\|\|F_1(s_0)$ |
| $S_B$ | 20 | 4 | $F_{10}(s_5)\|\|F_8(s_4)\|\|F_7(s_3)\|\|F_5(s_2)$ |
| $S_D$ | 20 | 4 | $F_2(s_1)\|\|F_0(s_0)\|\|F_{11}(s_5)\|\|F_9(s_4)$ |
| $S_P$ | 5 | 1 | $F_{13}(s_6)$ |
| $S_Q$ | 5 | 1 | $F_{12}(s_6)$ |

**Table 3.** Inputs from FSR-A into the S-box boolean functions

| Function input | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|---|
| $s_0$ | $A_{4,0}$ | $A_{1,2}$ | $A_{6,1}$ | $A_{7,3}$ | $A_{9,0}$ |
| $s_1$ | $A_{2,1}$ | $A_{3,2}$ | $A_{6,3}$ | $A_{7,0}$ | $A_{9,1}$ |
| $s_2$ | $A_{1,3}$ | $A_{2,0}$ | $A_{5,1}$ | $A_{5,3}$ | $A_{6,2}$ |
| $s_3$ | $A_{3,3}$ | $A_{1,1}$ | $A_{2,3}$ | $A_{4,2}$ | $A_{8,0}$ |
| $s_4$ | $A_{5,2}$ | $A_{4,3}$ | $A_{6,0}$ | $A_{8,1}$ | $A_{9,2}$ |
| $s_5$ | $A_{3,1}$ | $A_{4,1}$ | $A_{5,0}$ | $A_{7,2}$ | $A_{9,3}$ |
| $s_6$ | $A_{2,2}$ | $A_{3,0}$ | $A_{7,1}$ | $A_{8,2}$ | $A_{8,3}$ |

**Table 4.** Algebraic Normal Forms of 5-input Boolean functions $F_i, 0 \leq i < 14$

| i | Algebraic Normal Form of $F_i$ |
|---|---|
| 0 | $1 + x_4 + x_3 + x_3x_4 + x_2x_4 + x_2x_3 + x_1x_4 + x_1x_3 + x_1x_2 + x_1x_2x_4 + x_1x_2x_3 + x_0 + x_0x_3x_4 + x_0x_2 + x_0x_2x_3 + x_0x_1 + x_0x_1x_3 + x_0x_1x_3x_4 + x_0x_1x_2 + x_0x_1x_2x_3$ |
| 1 | $x_3 + x_2x_4 + x_1 + x_1x_4 + x_1x_3x_4 + x_0x_4 + x_0x_1 + x_0x_1x_3 + x_0x_1x_2 + x_0x_1x_2x_4$ |
| 2 | $1+x_4+x_3+x_2x_4+x_2x_3+x_2x_3x_4+x_1+x_0x_2x_3+x_0x_1x_4+x_0x_1x_3+x_0x_1x_3x_4+x_0x_1x_2$ |
| 3 | $1 + x_3 + x_2 + x_2x_4 + x_1x_3x_4 + x_1x_2x_4 + x_0x_3x_4 + x_0x_2 + x_0x_1 + x_0x_1x_3x_4 + x_0x_1x_2x_4$ |
| 4 | $1 + x_4 + x_3 + x_2x_4 + x_2x_3 + x_2x_3x_4 + x_1 + x_1x_4 + x_1x_3 + x_1x_3x_4 + x_1x_2 + x_1x_2x_3 + x_0 + x_0x_3 + x_0x_3x_4 + x_0x_2 + x_0x_2x_4 + x_0x_2x_3 + x_0x_2x_3x_4 + x_0x_1x_4 + x_0x_1x_2 + x_0x_1x_2x_3$ |
| 5 | $x_3 + x_3x_4 + x_2x_4 + x_1 + x_0$ |
| 6 | $1 + x_4 + x_3x_4 + x_2 + x_2x_3x_4 + x_1 + x_1x_2x_3 + x_0 + x_0x_4 + x_0x_3 + x_0x_2x_3x_4 + x_0x_1 + x_0x_1x_4 + x_0x_1x_3x_4 + x_0x_1x_2 + x_0x_1x_2x_3$ |
| 7 | $1 + x_3 + x_3x_4 + x_2 + x_1x_4 + x_1x_3x_4 + x_1x_2 + x_0x_4 + x_0x_3 + x_0x_2x_3x_4 + x_0x_1 + x_0x_1x_4 + x_0x_1x_3x_4 + x_0x_1x_2 + x_0x_1x_2x_3$ |
| 8 | $1+x_4+x_3+x_3x_4+x_2x_4+x_2x_3+x_2x_3x_4+x_1+x_1x_4+x_1x_3x_4+x_1x_2x_4+x_1x_2x_3+x_0x_4 + x_0x_3 + x_0x_2 + x_0x_2x_3 + x_0x_2x_3x_4 + x_0x_1x_4 + x_0x_1x_3 + x_0x_1x_2x_4 + x_0x_1x_2x_3$ |
| 9 | $x_3x_4 + x_2 + x_2x_4 + x_2x_3x_4 + x_1x_4 + x_1x_3 + x_1x_2x_4 + x_0x_4 + x_0x_2 + x_0x_2x_4 + x_0x_2x_3 + x_0x_2x_3x_4 + x_0x_1 + x_0x_1x_4 + x_0x_1x_3 + x_0x_1x_3x_4$ |
| 10 | $x_3 + x_2x_4 + x_1x_3x_4 + x_1x_2 + x_1x_2x_4 + x_0 + x_0x_3x_4 + x_0x_1x_4$ |
| 11 | $x_4+x_2+x_2x_3+x_2x_3x_4+x_1x_3+x_1x_2+x_1x_2x_3+x_0x_3x_4+x_0x_2x_3+x_0x_2x_3x_4+x_0x_1x_3x_4 + x_0x_1x_2x_3$ |
| 12 | $x_4+x_3+x_3x_4+x_2+x_1+x_1x_3x_4+x_0x_4+x_0x_3x_4+x_0x_2+x_0x_2x_3+x_0x_2x_3x_4+x_0x_1x_3x_4 + x_0x_1x_2x_3$ |
| 13 | $x_4 + x_3x_4 + x_2 + x_2x_3 + x_2x_3x_4 + x_1 + x_1x_2 + x_0 + x_0x_1x_3 + x_0x_1x_3x_4$ |