Three Snakes in One Hole: A 67 Gbps Flexible Hardware for SOSEMANUK with Optional Serpent and SNOW 2.0 Modes*

Goutam Paul
Department of Computer Science and Engineering,
Jadavpur University, Kolkata 700 032, India
goutam.paul@ieee.org

Anupam Chattopadhyay
Institute for Communication Technologies and Embedded Systems,
RWTH Aachen University, Aachen 52074, Germany
anupam.chattopadhyay@ice.rwth-aachen.de

Abstract

With increasing usage of hardware accelerators in modern heterogeneous System-on-Chips (SoCs), the distinction between hardware and software is no longer rigid. The domain of cryptography is no exception and efficient hardware design of so-called software ciphers are becoming increasingly popular. In this paper, for the first time we propose an efficient hardware accelerator design for SOSEMANUK, one of the finalists of the eSTREAM stream cipher competition in the software category. Since SOSEMANUK combines the design principles of the block cipher Serpent and the stream cipher SNOW 2.0, we make our design flexible to accommodate the option for independent execution of Serpent and SNOW 2.0. In the process, we identify interesting design points and explore different levels of optimizations. We perform a detailed experimental evaluation of the performance figures of each design point and in each case our figures by far outperform the existing benchmarks. The best throughput achieved by the combined design is 67.84 Gbps for SOSEMANUK, 33.92 Gbps for SNOW 2.0 and 2.12 Gbps for Serpent. The throughput for SOSEMANUK by far outperforms all existing benchmarks on the eSTREAM candidates.

Keywords: Cryptography, Hardware Accelerator, Serpent, SNOW 2.0, SOSEMANUK, Stream cipher implementation

1 Introduction

The eSTREAM [12] competition aimed at identifying modern stream ciphers in two separate profiles, one for software and the other for hardware platforms. Out of 34 initial submissions,

*This work was done in part while the first author was visiting RWTH Aachen, Germany as an Alexander von Humboldt Fellow.

four software stream ciphers, namely, HC-128, Rabbit, Salsa20/12, SOSEMANUK and three hardware stream ciphers, namely, Grain v1, MICKEY 2.0 and Trivium made into the final portfolio.

With advancement of technology, the difference between hardware and software stream ciphers is becoming blurred day by day. To satisfy the shrinking energy budgets, dedicated accelerators and customized instruction-sets are also commonly found in modern processors [3] and heterogeneous multiprocessor System-on-Chips (SoCs). Along the same direction, recent years have witnessed several attempts in hardware accelerator designs of software ciphers [20, 29, 40, 25, 18, 34, 30, 4, 32, 38].

In the call for the AES competition [1], one of the requirements was that the cipher should be implementable in both hardware and software. After Rijndael [9] won the competition in 2001, initial few years were predominated by software implementations. However, subsequently many hardware designs have been attempted and now Intel has made a special AES instruction set in their x86 series of processors [3].

The story of eSTREAM competition [12] is however different. It created two separate profiles for software and hardware. Some of the initial submissions, such as Rabbit and Salsa20/12, were for both the profiles. During later rounds the categorization was made exclusive and both Rabbit and Salsa20/12 were moved to the software category. From submission to final selection, SOSEMANUK [6] was in the software category all throughout. However, in [16], hardware performances of selected eSTREAM candidates were analyzed and the following interesting conclusion was drawn about SOSEMANUK.

> With regard to SOSEMANUK, the utility as a hardware cipher is clear thus in our opinion requires adding to the hardware focus profile.

However, it is surprising that no hardware design was attempted for SOSEMANUK after [16] which remains the only hardware benchmark for this cipher so far. This is despite the fact that there is no practical attack on SOSEMANUK and the cipher retains its claimed 128-bit security. There exist hardware designs for the other eSTREAM software finalists, e.g., for HC-128 [4], Rabbit [34] and Salsa20/12 [40, 18]. In this paper, we complete the picture by proposing an efficient hardware for SOSEMANUK. We design a flexible accelerator for SOSEMANUK with additional modes for Serpent [5] block cipher and SNOW 2.0 stream cipher [11] whose design principles are used to construct SOSEMANUK.

The origin of the name SOSEMANUK is explained in [6, Section 1]. Literally, it means snow-snake, which is appropriate since it combines Serpent (which literally means snake) and SNOW 2.0. Though the word *snow* does not imply any kind of snake (except possibly snake-shaped object made out of snow), we note that the names of all the three ciphers begin with the letter "S" which is itself serpentine in shape! Hence we take the liberty to refer to the three ciphers as three snakes in the title.

1.1 Our Contributions

We list our contributions as follows.

1. We propose a novel and efficient hardware for SOSEMANUK. It is the first of its kind since other than [16], no other hardware design of SOSEMANUK has been attempted.

2. For the first time, we present a flexible accelerator that combines a block cipher and two stream ciphers.

3. We identify 12 incremental design points in the design process and report optimizations and evaluations of each of them.

4. Our design outperforms all existing hardware (as well as software) designs of Serpent, SNOW 2.0 and SOSEMANUK, along with those of all other eSTREAM candidates.

5. We propose a tweak to prevent the differential fault attacks [28, 24] on SOSEMANUK with negligible increase in area and no compromise on throughput.

6. Duplicating hardware components to perform parallel data stream processing for throughput maximization is done in some of the existing cryptographic hardware designs[3]. We do not employ such tricks and apart from absolute throughput and area, we also report throughput per area as one of the figures of merit.

2 Brief Description of Serpent, SNOW 2.0 and SOSEMANUK

SOSEMANUK [6] combines the design philosophies of the block cipher Serpent [5] and the stream cipher SNOW 2.0 [11]. Below we mention the salient design features of each of the three ciphers.

2.1 Description of Serpent

Serpent was a candidate for the AES competition. It is a 32-round SP-network operating on four 32-bit words. It encrypts a 128-bit plaintext P to a 128-bit ciphertext C in 32 rounds under 33 many 128-bit subkeys $\hat{K}_0, \ldots, \hat{K}_{33}$. The cipher supports three different key lengths, namely 128-bit, 192-bit or 256-bit. Keys with less than 256 bits are expanded into full 256-bit keys by appending one "1" bit to the MSB end, followed by as many "0" bits as required. Serpent uses 8 many 4-to-4-bit S-boxes $S_0, \ldots, S_7$. The cipher can be formally described as

$$\hat{B}_0 = IP(P), \qquad \hat{B}_{i+1} = R_i(\hat{B}_i), \qquad C = FP(\hat{B}_{32}),$$

where IP and FP are the initial and the final permutations respectively over the 128 bit-positions and the round function R_i is defined as

$$R_i(X) = L\left(\hat{S}_i(X \oplus \hat{K}_i\right), \text{ for } i = 0, \ldots, 30,$$

$$R_i(X) = \hat{S}_i\left(X \oplus \hat{K}_i\right) \oplus \hat{K}_{32}, \text{ for } i = 31.$$

Here L is a linear transformation (LT) and $\hat{S}_i$ is the application of the S-box $S_{i \bmod 8}$ 32 times in parallel. When the S-boxes are applied in bitslice mode, each of them act as 128-bit to 128-bit S-box and the initial and final permutation steps are no longer required. We describe the S-boxes in Appendix A.1 and the linear transformation in Appendix A.2.

The 256-bit effective key (after necessary padding) is written as eight 32-bit words w_{-8}, ..., w_{-1} which are then expanded into an intermediate key $w_0, \ldots, w_{132}$ by the following recurrence.

$$w_i = (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11, \tag{1}$$

for $i = 0, \ldots, 131$, where ϕ is the fractional part of the golden ratio $(\sqrt{5}+1)/2$ or $0x9e3779b9$ in hexadecimal. Now the S-boxes are used to transform the prekeys w_i into words k_i of the round keys. This transformation is described in Appendix A.3.

2.2 Description of SNOW 2.0

SNOW 2.0 [11] uses an LFSR of length 16 (each entry is a 32-bit word) with feedback polynomial

$$\pi(x) = \alpha x^{16} + x^{14} + \alpha^{-1}x^5 + 1 \in \mathbb{F}_{2^{32}}[x],$$

where α is a root of $x^4 + \beta^{23}x^3 + \beta^{245}x^2 + \beta^{48}x + \beta^{239} \in \mathbb{F}_{2^8}[x]$ and β is a root of $x^8 + x^7 + x^5 + x^3 + 1 \in \mathbb{F}_2[x]$.

Let $(s_{t+15}, \ldots, s_t)$ denote the state of the LFSR at time $t \geq 0$. There is a finite state machine (FSM) with two registers $R1, R2$ and an S-box S. The output of the FSM and the keystream word generated are respectively given by

$$F_t = (s_{t+15} \boxplus R1_t) \oplus R2_t, \text{ for } t \geq 0,$$

$$z_t = F_t \oplus s_t, \text{ for } t \geq 1.$$

For $t \geq 0$, the registers R1 and R2 are updated as

$$R1_{t+1} = s_{t+5} \boxplus R2_t, \qquad R2_{t+1} = S(R1_t).$$

SNOW 2.0 supports a secret key K of either 128 or 256 bits and a 128-bit initialization vector (IV). The details of the key initialization are given in Appendix B.

2.3 Description of SOSEMANUK

SOSEMANUK [6] has a key length of either 128 bits or 256 bits and an IV of 128 bits. It uses two primitives from Serpent, namely, *Serpent24* used in the key schedule and *Serpent1*, used during the keystream generation. *Serpent24* is Serpent reduced to 24 rounds instead of 32 rounds, where the last round (i.e., 24th round) retains the linear transformation unlike true Serpent. Thus,

$$R_{23}(X) = L\left(\hat{S}_{23}(X \oplus \hat{K}_{23})\right) \oplus \hat{K}_{24}.$$

Serpent1 is just one round of Serpent with the S-box S_2, but without the key addition and the linear transformation.

The LFSR used is defined over the same finite field as in SNOW 2.0, but is of length 10 instead of 16. The new value is computed as

$$S_{t+10} = S_{t+9} \oplus \alpha^{-1}s_{t+3} \oplus \alpha s_t, \qquad t \geq 1.$$

The FSM uses two 32-bit registers $R1$, $R2$ as in SNOW 2.0, but instead of an S-box connecting them, it has a transformation $Trans$ connecting them. The update of the FSM for $t \geq 1$ and the output f_t are given below.

$$\begin{aligned} R1_t &= R2_{t-1} + mux(lsb(R1_{t-1}, s_{t+1}, s_{t+1} \oplus s_{t+8})) \bmod 2^{32}, \\ R2_t &= Trans(R1_{t-1}), f_t = (s_{t+9} + R1_t \bmod 2^{32}) \oplus R2_t, \end{aligned}$$

where $lsb(x)$ is the least significant bit of the word x and $mux(c, x, y)$ selects x if $c = 0$, or y if $c = 1$, and

$$Trans(z) = (0x54655307 \times z \bmod 2^{32}) \lll 7.$$

The outputs of the FSM are grouped by four and then the output keystream words are generated as

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = Serpent1(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t).$$

The key setup corresponds to the key setup of *Serpent24*, that produces 25 128-bit subkeys. The 128-bit IV is used as input to *Serpent24* block cipher and the outputs $(Y_3^r, Y_2^r, Y_1^r, Y_0^r)$ from the r-th rounds of *Serpent24*, corresponding to $r = 12, 18, 24$, are used to load $R1$, $R2$ as $(R1, R2) = (Y_0^{18}, Y_2^{18})$ and the LFSR as

$$\begin{aligned}(s_6, s_7, s_8, s_9) &= (Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12}),\\ (s_4, s_5) &= (Y_1^{18}, Y_3^{18}),\\ (s_0, s_1, s_2, s_3) &= (Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24}).\end{aligned}$$

3 Design Space Exploration

We started with the design and related optimizations in an incremental fashion and the process led to 12 different design points. For ease of discussion, let us introduce a few notations. Let S_e, S_n and S_o denote the implementations of the individual ciphers Serpent, SNOW 2.0 and SOSEMANUK respectively. Analogously, S_{en} means a combined implementation for both Serpent and SNOW 2.0 and S_{eno} means a combined implementation of all the three ciphers. We use a second subscript u preceded by a comma to denote a version of the same cipher with the LFSR unrolled (we would explain shortly what does unrolled mean). We use a superscript (n) to denote that there are n pipeline stages in the design. For example, $S_{eo,u}^{(2)}$ means a 2-stage implementation of SOSEMANUK and Serpent together with the LFSR unrolled.

Table 1: Division of Serpent and LFSR operations into different pipeline stages

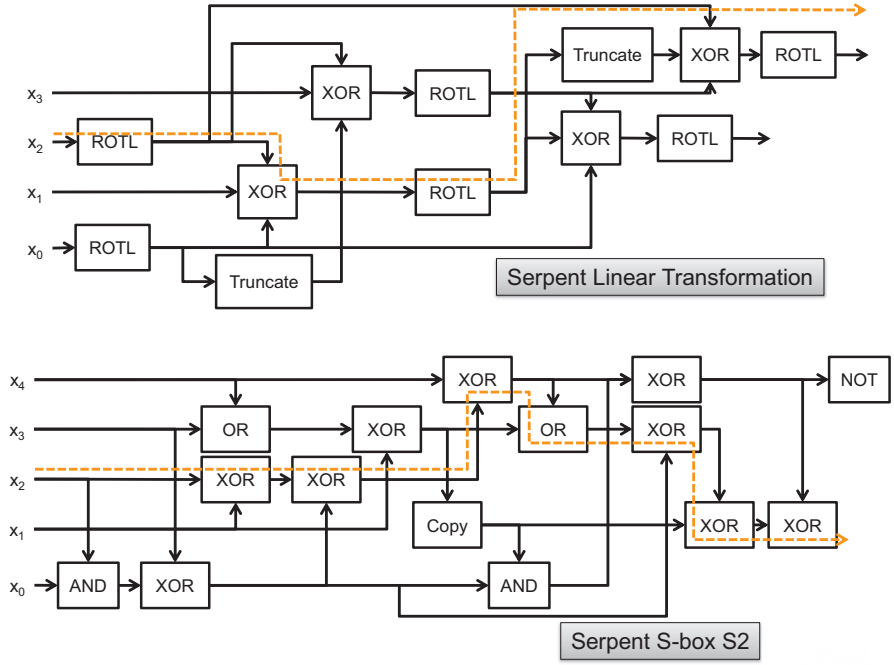
Serpent		LFSR operations	
Stage 1	Stage 2	Stage 1	Stage 2
Computation of recurrence specified in Equation 1		Address generation for α and α^{-1} accesses	Read α and α^{-1}
Address generation for round key	Read/Write round key		Update R1, R2, Update and shift LFSR, and Keystream generation
Load input for S-box computation	S-box computation, Linear transformation, and Keystream generation		

The LFSR evolution of SNOW 2.0 and SOSEMANUK is always implemented in 2 pipeline stages. Whereas the Serpent (in S_e, S_{en}, S_{eo} or S_{eno}) is implemented in either 2 stages or 3 stages. In Table 3, we show how the Serpent components were divided across the different pipeline stages for 2-stage implementation. *In case of the 3-stage implementation, the linear transformation is done in the 3rd pipeline stage and the loading of input operands for LT is done in the 2nd state.*

The critical paths for S-box and LT are shown in dotted lines in Fig. 1, justifying their separation across two stages. The splitting of S-box computation and linear transformation between two pipeline stages incurs a throughput degradation from 1 round per cycle to 1 round per 2 cycles. On the other hand, this decision increases the maximum achievable clock frequency.

In Table 3, we show how the LFSR components were divided across the different pipeline stages. The rationale for generating the addresses in the first pipeline stage is twofold. First, it accommodates for the 1 cycle read latency of the storage. Second, the addresses for the next SNOW 2.0 iteration is already available in the LFSR and therefore, the pipeline can operate at maximum throughput. Splitting the second pipeline stage into further stages would either cause a decrease in the throughput or a complex bypass logic leading to the same critical path.

Figure 1: Critical paths for Serpent Round functions



We started with a basic design of 3-stage Serpent, which we call $S^{(3)}_{e,basic}$. Both from timing and area perspective, several optimizations were applied to this design point leading to an optimized version $S^{(3)}_{e}$. The optimizations are explained in Section 5. These optimizations were retained in subsequent evolution of the design points. From $S^{(3)}_{e}$, we reduced one pipeline stage to double the throughput of Serpent, in terms of bits per cycle, yielding $S^{(2)}_{e}$. Now we added a 2-stage version of SNOW 2.0 onto it, giving us $S^{(2)}_{en}$. From this, we created an independent version of SNOW 2.0, i.e., $S^{(2)}_{n}$. From $S^{(2)}_{en}$, we gradually developed $S^{(2)}_{eno}$, $S^{(3)}_{eno}$ and $S^{(3)}_{eno,u}$, one after another. We did an experiment with further SNOW 2.0 unrolling here and created $S^{(3)}_{eno,uu}$. To keep the focus on Sosemanuk, we traced back to $S^{(3)}_{eno,u}$ and then we bifurcated - in one path we developed $S^{(3)}_{eo,u}$ and in another path we developed first $S^{(2)}_{eno,u}$ and then $S^{(2)}_{eo,u}$. Finally, we developed a fault attack resistant version of $S^{(2)}_{eno,u}$, denoted by $S^{\prime(2)}_{eno,u}$. All the design iterations were guided by intermediate performance evaluation of the RTL description.

4 Instruction Set Design

For different modes of operation in the architecture, different instructions are designed. This allows a convenient assembly-level programming environment as well as limited algorithmic flexibility.

4.1 Serpent Rounds

For Serpent key scheduling, eight 32-bit words, namely, $w_0, \ldots, w_7$ are operated with the same transformation, however, with different variable ordering. To keep the operator generic, the variable ordering is encoded in the instruction. For 4 different variable ordering, 4 different instructions are designed. Each bitsliced implementation of Serpent S-box is triggered via one specific instruction. This requires total 4 + 8, i.e., 12 instructions.

Each of the Serpent bitsliced S-boxes takes five inputs, of which the first four contain four 32-bit words to process and the fifth one serves as an auxiliary variable. We use five variables, denoted by r_i, $i = 0, \ldots, 4$, for the S-box and the linear transformation (LT). If the inputs to the S-box are in r_0, r_1, r_2, r_3, then r_4 is the default auxiliary variable and as per the S-box definitions, the output indices are given in the second column of Table 4.1. These S-box outputs go directly as inputs to LT, which produces the outputs in the same locations.

Table 2: Input and output indices of r_i for different Serpent rounds

S-box	Standard mapping (when S-box in = 0, 1, 2, 3) S-box out = LT in	In first 8 rounds		
		S-box in	S-box out (as per standard mapping)	LT in (as per our permutation network)
S_0	1, 4, 2, 0	0, 1, 2, 3	1, 4, 2, 0	1, 4, 2, 0
S_1	2, 0, 3, 1	1, 4, 2, 0	2, 1, 0, 4	2, 0, 3, 1
S_2	2, 3, 1, 4	2, 1, 0, 4	0, 4, 1, 3	2, 3, 1, 4
S_3	1, 2, 3, 4	0, 4, 1, 3	4, 1, 3, 2	1, 2, 3, 4
S_4	1, 4, 0, 3	4, 1, 3, 2	1, 0, 4, 2	1, 4, 0, 3
S_5	1, 3, 0, 2	1, 0, 4, 2	0, 2, 1, 4	1, 3, 0, 2
S_6	0, 1, 4, 2	0, 2, 1, 4	0, 2, 3, 1	0, 1, 4, 2
S_7	4, 3, 1, 0	0, 2, 3, 1	4, 1, 2, 0	4, 3, 1, 0

However, the outputs of LT need to be fed as input to the next S-box in the next round. Thus, after the first round, S_0 puts the outputs in r_1, r_4, r_2, r_0 which also remain the outputs of LT. In the second round, S_1 takes inputs from r_1, r_4, r_2, r_0 and produces outputs in r_2, r_1, r_0, r_4 (as per row S_1, column 4 in the table) instead of r_2, r_0, r_3, r_1 (row S_1, column 2 in the table). This continues and the indices for the first 8 rounds are shown in the third and fourth column of Table 4.1.

In software, the S-box and LT are implemented typically as functions or macro and therefore in any serpent round the S-box output indices and the LT input indices can remain the same. On the other hand, the default hardware implementation is to use array-based signals [16] for passing the data between round key access, S-box computation and linear transformation. Unlike [16], we created a software-controlled permutation network. The inputs and outputs of the permutation networks are shown in the fourth and the fifth column of Table 4.1.

The mapping of the permutation network can be explained by an example as follows. Consider the 5th Serpent round, i.e., the row corresponding to S_4 in the table. The indices for the S-box input are $4, 1, 3, 2$ and those for the S-box output are $1, 0, 4, 2$. If one creates a list $[4, 1, 3, 2, 0]$ of the input r_i indices, where position 4 corresponds to r_0, then the positions of the output indices $1, 0, 4, 2$ in this list is given by $1, 4, 0, 3$ respectively. As shown in the table, this is precisely the output of the permutation network, which serves as the input to the next LT.

In the accelerator design exploration, this assembly control of permutation network indices allowed us to efficiently implement round key access, S-box implementation and linear transformation. The permutation network is decoupled from the combinatorial logic, which could be conveniently moved between pipeline stages for best timing results. A subtle benefit of this scheme is the possibility to accommodate different permutation network mapping for different algorithm variants.

For each of these Serpent round functions, at least 4 indices are required, where the 5th index can be computed from them. This requires total 12 3-bit indices requiring total 36 bits. To restrict the instruction bitwidth within 32, an instruction is issued before the first round specifying the input indices for round key function. The input indices of linear transformation of round n acts as input indices of S-box of round $n + 1$, which are stored in registers.

For every round, the same instruction with different index parameters is called. There is a special instruction for the final round, which skips the linear transformation. Therefore, total 3 different instructions for initialization, Serpent round and final round are needed.

The instruction set is flexible for diverse indexing options in the Serpent rounds as well as different order of S-box accesses during key scheduling. Naturally, the increase or decrease of Serpent rounds is also possible.

4.2 SNOW 2.0 Operations

The instruction set for SNOW 2.0 contains only 3 instructions namely, load key, initialization and keystream generation. The key, IV and keylength are loaded via input pins. This is followed by 32 rounds of initialization. Finally the keystream generation instruction is issued. Naturally, the datapath for initialization and keystream generation is shared.

The LFSR feedback polynomial is hardwired in the microarchitecture for maximizing the performance.

4.3 Additional SOSEMANUK Operations

The key scheduling and round functions' instructions from Serpent could be completely reused for SOSEMANUK. Additionally, SOSEMANUK initialization required one instruction for loading the LFSR. This instruction requires two different sets of parameters. The first set specifies indices of input data and the second set specifies LFSR indices. Since we stored the output indices of every Serpent round, the first set of parameters are already available in the microarchitecture. Therefore, only the LFSR indices need to be stored. The values are stored into $R1$ and $R2$, when the LFSR indices are specified as 10 and 11 respectively. Note that, SNOW 2.0 uses a larger LFSR compared to SOSEMANUK leaving few LFSR positions redundant.

For the encryption operation, two specific instructions for keystream generation and Serpent round call is designed. The keystream generation for SOSEMANUK uses a different transformation compared to SNOW 2.0 though, the rest of the datapath is shared.

All the instructions are 32 bit wide, of which 2 bits are used to distinguish between different *mode* of the application. Currently, three different modes i.e. Serpent, SNOW 2.0 and SOSEMANUK are supported. Depending on the mode, slightly different behavior for LFSR shifting, register initialization and S-box access is triggered.

5 Microarchitecture Design and Optimizations

We describe the different design choices and optimizations of the microarchitecture in the following subsections.

5.1 Storage

There are three specific requirements for storage among Serpent, SNOW 2.0 and SOSEMANUK.

For SNOW 2.0 and SOSEMANUK, α and α^{-1} values are precomputed and stored in 256 entry 32-bit wide look-up tables. For initial implementation of SOSEMANUK, 1 read port is sufficient for both the tables. For unrolled version, 2 read ports are required for each.

For Serpent round key, 132 entry 32-bit wide storage with both read and write operations is required. Since each Serpent round requires 4 accesses to the storage, it is divided into two separate memories storing even and odd locations. Each memory is designed with 2 combined ports allowing 4 simultaneous accesses.

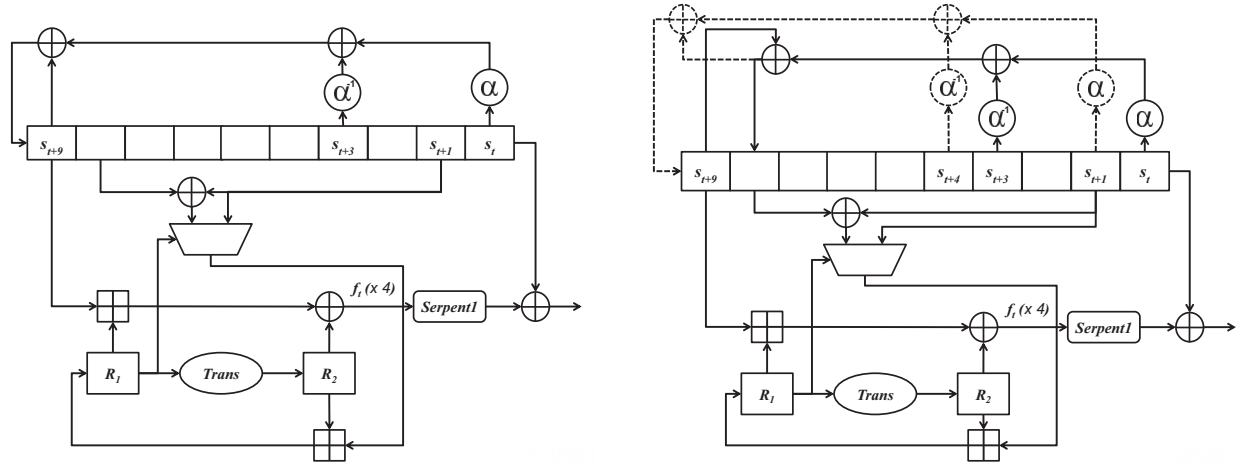
SNOW 2.0 requires an S-box implementation with 32-bit input and 32-bit output. This S-box can be decomposed into the 8-bit input, 8-bit output Rijndael S-box and a few logical operations [11, Section 6]. The complete Rijndael S-box is hardcoded into the architecture, which incurs little area overhead and does not affect the runtime performance.

5.2 Sliding LFSR

SNOW 2.0 has 16 32-bit registers and SOSEMANUK requires only 10 32-bit registers. In our generic design we have 16 32-bit registers and hence in SOSEMANUK mode, 6 slots of the LFSR are free. For SOSEMANUK keystream generation, four consecutively dropped words from the LFSR are XOR-ed with four consecutive *Serpent1* outputs. Instead of storing the dropped words in four separate registers, we use the LFSR locations 4 to 13 to implement the LFSR of SOSEMANUK and use the locations 0 to 3 to store the dropped words before they are XOR-ed. Thus, the computation of storing these words is automatically consumed in the natural LFSR update.

5.3 Unrolled LFSR

Figure 2: Schematic Diagram of LFSR Unrolling



(a) LFSR Rolled

(b) LFSR Unrolled

For some of our design points, we create a version with the LFSR unrolled for two steps with an aim to achieving better throughput. The idea is to perform two consecutive updates of the LFSR in one clock cycle. This involves shifting of the LFSR by two positions and loading the positions S_{t+9} and S_{t+8} with two new values. Pictorially the rolled and the unrolled versions of the LFSR are shown in Fig. 2. For clarity, the unrolling effects in the FSM update is not shown. Naturally, it involves two consecutive computations of $R1$ and $R2$ in the same clock cycle.

In principle, further unrolling is possible. However, the *Serpent1* function for keystream generation is called after every four iterations of LFSR updates of SOSEMANUK. By unrolling

two steps of output generation, the *Serpent1* function needs to be called once after every 2 cycles (4/2) of our implementation. If we unroll one more iteration, it would mean that the *Serpent1* function needs to be called after every $\lceil 4/3 \rceil = 2$ cycles. In other words, we need to wait till 2 cycles of our implementation anyway before generating the keyword and this gives us no advantage at all.

5.4 Additional Optimizations

Apart from LFSR unrolling and optimization of the permutation network, several other design optimizations are performed to improve the throughput and area. These are briefly described in the following.

- The rotate operations in Serpent contain constant operands. Instead of having a flexible rotation unit, dedicated bit wiring is used for the rotations.
- Each of the 8 S-boxes in Serpent are accessed in a particular order. An 8-bit global register, called *serpent_rk_index*, is used for incrementing the indices of round key access. The 5 lower-order bits from the same register are used to determine the particular S-box to be called in a particular round.
- Serpent key scheduling requires 8 registers, namely, $w_0, \ldots, w_7$. These are re-used again during keystream generation for two different purposes. First, for storing the S-box input indices for the next Serpent round. Second, for holding the intermediate values $f_{t+3}, f_{t+2}, f_{t+1}, f_t$ of SOSEMANUK during the generation of its keystream.

5.5 Security Enhancement

Most of the attacks on SNOW 2.0 and SOSEMANUK have complexity more than 2^{128} and hence not practical for a keylength of 128 bits. These works include the guess and determine attacks in [2, 39, 15] and linear cryptanalysis of SOSEMANUK and SNOW 2.0 [21, 8].

There are two fault attacks on SOSEMANUK with better complexity. The differential fault attack of [28] requires around 6144 faults, and this is an work equivalent to around 2^{48} SOSEMANUK iterations and a storage of around $2^{38.17}$ bytes. The time complexity of 2^{48} is dominated by a pruned complexity of 2^{16} to guess the values of 8 LFSR states and 8 FSM outputs and a complexity of 2^{32} to guess the initial value of $R1$. In [24], an improved attack is presented that requires only around 4608 faults, $2^{35.16}$ SOSEMANUK iterations and $2^{23.46}$ bytes storage. This complexity is dominated by $2^{3.16}$ (instead of 2^{16} as in [28]) for the first part and a complexity of 2^{32} to guess the initial value of $R1$.

To prevent this fault attack, we duplicate the LFSR's S_1, S_8 and S_9. We call this variant $S'^{(2)}_{eno,u}$. If at any step the two copies of any one of the three LFSR's do not have the same value, then the process is aborted. Thus the complexity of guessing the LFSR states is increased by at least 2^{96}, thereby moving the total complexity beyond 2^{128}.

6 Performance Evaluation

All the design points were first modeled in *Synopsys Processor Designer* version G-2012.06-SP2 Linux [37], a high-level processor design environment. The algorithm outputs were verified with cycle-accurate instruction-set simulation. Optimized RTL implementation is

generated from the high-level description automatically, which is again functionally verified by running RTL simulation. The high-level design environment considerably reduced the modeling and exploration efforts. The RTL model complexity, in terms of lines of code, is approximately 20× that of the high-level description.

The generated RTL model is synthesized with *Synopsys Design Compiler* version D-2010.03-SP4 [35], with target 65nm ASIC technology library. During synthesis, *compile_ultra* option with high timing effort and topographical mode is used. Repeated synthesis with increasing clock frequency is performed as long as no timing violation is reported. The generated timing results are used to analyse the critical path and then timing optimizations to the high level description of the model are applied accordingly. RTL switching activity is recorded and provided as an input to *Synopsys PrimeTime* version D-2010.03-SP4 [36]. for obtaining power estimates. The performance estimates for memory structures are obtained by using *Faraday Memory Compiler* [14], 65nm technology library. For all the design points, the memory access time satisfies the core frequency.

6.1 Area, Timing and Power

The evolution of design points are associated with corresponding area, timing and power figures. For convenience, first the area results are presented in Table 3, followed by throughput, power and energy-efficiency results in Table 4. In the following, the design evolution, its rationale and the observed results are presented stepwise.

Table 3: Design Area Distribution

Design	Core Area (KGates)			Memory			Total Area
	Combinational	Sequential	Total	Ports	(Bytes)	(KGates)	(KGates)
$S^{(3)}_{e,basic}$	41.391	4.585	45.976	2 combined	528	15.84	61.816
$S^{(3)}_{e}$	30.595	4.689	35.284	2 combined	528	15.84	51.124
$S^{(2)}_{e}$	39.989	3.67	43.659	2 combined	528	15.84	59.499
$S^{(2)}_{en}$	42.711	6.848	49.559	2 combined, 1 read	528, 2048	32	81.559
$S^{(2)}_{n}$	10.796	3.7	14.496	1 read	2048	16.24	30.736
$S^{(2)}_{eno}$	53.083	7.031	60.114	2 combined, 1 read	528, 2048	32	92.114
$S^{(3)}_{eno}$	53.728	7.976	61.704	2 combined, 1 read	528, 2048	32	93.704
$S^{(3)}_{eno,u}$	**49.334**	**7.835**	**57.169**	**2 combined, 2 read**	**528, 2048**	**45.8**	**102.969**
$S^{(3)}_{eno,uu}$	69.308	7.973	77.281	2 combined, 2 read	528, 2048	45.8	123.08
$S^{(3)}_{eo,u}$	42.13	7.81	49.94	2 combined, 2 read	528, 2048	45.8	95.74
$S^{(2)}_{eo,u}$	56.602	6.994	63.596	2 combined, 2 read	528, 2048	45.8	109.396
$S^{(2)}_{eno,u}$	**67.801**	**7.071**	**74.872**	**2 combined, 2 read**	**528, 2048**	**45.8**	**120.672**
$S'^{(2)}_{eno,u}$	67.743	7.598	75.341	2 combined, 2 read	528, 2048	45.8	121.141

$S^{(3)}_{e,basic} \rightarrow S^{(3)}_{e}$: The basic Serpent implementation, i.e., $S^{(3)}_{e,basic}$, is improved with the permutation network optimization of Section 4.1 and other optimizations mentioned in Section 5.4 to significantly improve the area. The introduction of instruction-set and corresponding decoding logic compromised the throughput slightly.

$S^{(3)}_{e} \rightarrow S^{(2)}_{e}$: In order to achieve higher bits per cycle, we moved to a 2-stage Serpent implementation.

$S^{(2)}_{e} \rightarrow S^{(2)}_{en}$: SNOW 2.0 is included in the design, and this results in further area increase, mainly contributed by the memories storing α and α^{-1} values.

$S_{en}^{(2)} \rightarrow S_{n}^{(2)}$: To gauge the achievable runtime performance of SNOW 2.0, a standalone implementation is next attempted. This also shows that the combined implementation of Serpent and SNOW 2.0, $S_{en}^{(2)}$ consumes less area than simple addition of individual SNOW 2.0 and Serpent implementations. This reflects and efficient sharing of resources.

$S_{en}^{(2)} \rightarrow S_{eno}^{(2)}$: SOSEMANUK mode is included in $S_{en}^{(2)}$ to reach $S_{eno}^{(2)}$. This caused an increase in the area by 10.55 KGates. The achievable throughput is constrained by the Serpent critical path at this design point.

$S_{eno}^{(2)} \rightarrow S_{eno}^{(3)}$: To improve the throughput of SOSEMANUK, Serpent datapath is distributed among 3 pipeline stages. This resulted in minor area increase, reduced Serpent throughput and increased SOSEMANUK's initialization latency. However, the clock frequency improved from 1010 MHz to 1280 MHz, improving the keystream generation speed of both SNOW 2.0 and SOSEMANUK.

$S_{eno}^{(3)} \rightarrow S_{eno,u}^{(3)}$: Recognizing that the critical path of Serpent round with the permutation network is able to accommodate larger combinational path of SOSEMANUK, we decided to unroll the LFSR. The unrolling affected the FSM update of SOSEMANUK. Though it reduced the clock frequency from 1280 MHz to 1060 MHz, it increased the throughput significantly by doubling the bits per cycle. Corresponding area increase is nominal, which is also reflected in the throughput per area improvement in Fig. 3 between $S_{eno}^{(3)}$ and $S_{eo,u}^{(3)}$.

$S_{eno,u}^{(3)} \rightarrow S_{eno,uu}^{(3)}$: We attempted further unrolling for the SNOW 2.0. It reduced the clock frequency even further and increased the area significantly. This caused a sharp drop in the throughput per area for SOSEMANUK. Naturally, the same metric is improved for SNOW 2.0.

$S_{eno,u}^{(3)} \rightarrow S_{eo,u}^{(3)}$: The area, timing and power results for SOSEMANUK implementation without SNOW 2.0 mode is studied at this point. Omission of SNOW 2.0 mode reduced the area by only 7.229 KGates, which is 7% of the total area. This is understandable, since the area overhead of SNOW 2.0 is dominated by α and α^{-1} values which are present in SOSEMANUK anyway.

$S_{eo,u}^{(3)} \rightarrow S_{eo,u}^{(2)} \rightarrow S_{eno,u}^{(2)}$: We moved back again from a 3-stage to a 2-stage implementation. The rationale is that the critical path of unrolled SOSEMANUK datapath is comparable to the critical path of a 2-stage Serpent implementation. Therefore, it is advisable to retain the 2-stage Serpent implementation for higher Serpent throughput. This reduces the throughput of SNOW 2.0 and SOSEMANUK slightly at the benefit of increased Serpent throughput ($S_{eno,u}^{(2)}$).

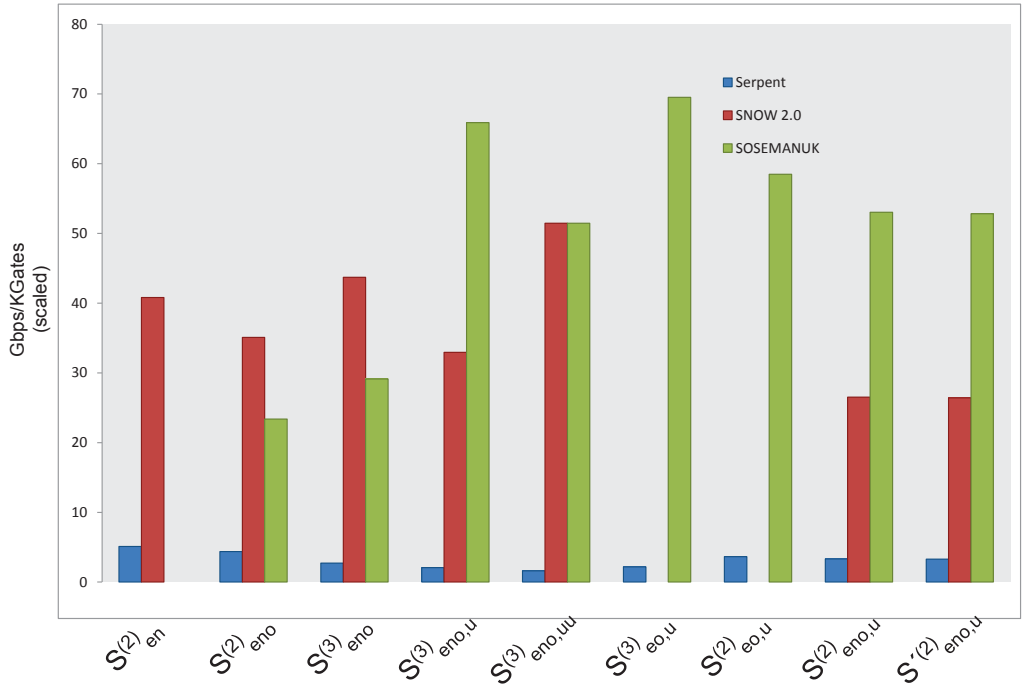
$S_{eno,u}^{(2)} \rightarrow S_{eno,u}^{\prime(2)}$: Finally, fault detection logic is implemented, which does not affect the throughput at all. The area increment is only 0.469 KGates.

From Table 4, the variation of throughput along the design points can be observed. For Serpent, a move from 3-stage to 2-stage implementation is always associated with an increase in throughput. The implementation of SNOW 2.0 is done for all the design points in 2 pipeline stages, resulting in 32 bits per cycle throughput. For SOSEMANUK, the initial implementation at $S_{eno}^{(2)}$ and $S_{eno}^{(3)}$ generated 128 bits of output after every 6 cycles. This is due to 4 consecutive LFSR operations followed by a 1-cycle stall when the intermediate values are

Table 4: Design Runtime, Energy Performance

Design	Core Frequency (MHz)	Throughput (bits per cycle)			Throughput (Gbps)			Core Power (mW)			Energy (pJ/bit) Core			Energy (pJ/bit) Memory		
		SE	SN	SO	SE	SN	SO	SE	SN	SO	SE	SN	SO	SE	SN	SO
$S_{e,basic}^{(3)}$	1600	2	-	–	3.2	–	–	37.73	–	–	11.89	–	–	8.11	–	–
$S_{e}^{(3)}$	1500	2	-	–	3	–	–	27.39	–	–	9.13	–	–	8.11	–	–
$S_{e}^{(2)}$	1060	4	-	–	4.24	–	–	37.47	–	–	8.84	–	–	8.11	–	–
$S_{en}^{(2)}$	1040	4	32	–	4.16	33.28	–	28.65	10.75	–	6.89	0.32	–	8.11	0.58	–
$S_{n}^{(2)}$	1950	-	32	–	–	62.4	–	–	11.24	–	–	0.18	–	–	0.58	–
$S_{eno}^{(2)}$	1010	4	32	21.33	4.04	32.32	21.54	28.23	16.34	25	6.99	0.51	1.16	8.11	0.58	0.58
$S_{eno}^{(3)}$	1280	2	32	21.33	2.56	40.96	27.3	28.25	18.43	24.44	11.04	0.45	0.9	8.11	0.58	0.58
$S_{eno,u}^{(3)}$	**1060**	**2**	**32**	**64**	**2.12**	**33.92**	**67.84**	**20.39**	**21.29**	**20.40**	**9.62**	**0.63**	**0.3**	**8.11**	**0.53**	**0.53**
$S_{eno,uu}^{(3)}$	990	2	64	64	1.98	63.36	63.36	24.78	48.99	25.40	12.51	0.77	0.4	8.11	0.53	0.53
$S_{eo,u}^{(3)}$	1040	2	-	64	2.08	–	66.56	20.31	–	19.67	9.76	–	0.3	8.11	–	0.53
$S_{eo,u}^{(2)}$	1000	4	-	64	4	–	64	29.01	–	27.81	7.25	–	0.43	8.11	–	0.53
$S_{eno,u}^{(2)}$	**1000**	**4**	**32**	**64**	**4**	**32**	**64**	**31.84**	**19.9**	**30.81**	**7.96**	**0.62**	**0.48**	**8.11**	**0.53**	**0.53**
$S_{eno,u}^{\prime(2)}$	1000	4	32	64	4	32	64	34.56	20.49	33.85	8.64	0.64	0.53	8.11	0.53	0.53

Figure 3: Area Efficiency Chart for Multi-mode Designs



loaded in the Serpent1 address generation instruction. In the 6th cycle, the *Serpent1* function is executed. $S^{(3)}_{eno}$ onwards the LFSR was unrolled. This with parallel execution of *Serpent1* function and LFSR operations resulted in a throughput of 64 bits per cycle. The decision of moving from 3-stage to 2-stage implementation ($S^{(3)}_{eno,u} \rightarrow S^{(2)}_{eno,u}$) reduces the achievable throughput for SNOW 2.0 and SOSEMANUK at the cost of increased Serpent throughput.

For the multi-mode design points, the area efficiency results in terms of throughput per area are presented in Fig. 3. The gradual changes in the area efficiency between different points are as following.

$S^{(2)}_{en} \rightarrow S^{(2)}_{eno}$: The area efficiency of both SNOW 2.0 and SOSEMANUK decrease for accommodating an additional mode.

$S^{(2)}_{eno} \rightarrow S^{(3)}_{eno}$: By increasing the pipeline stages, higher throughput with little area increase is achieved for SNOW 2.0 and SOSEMANUK. Area efficiency for Serpent decreases.

$S^{(3)}_{eno} \rightarrow S^{(3)}_{eno,u}$: LFSR unrolling improves the throughput of SOSEMANUK, though increasing the overall area. This is reflected in reduced area efficiency for SNOW 2.0 and Serpent.

$S^{(3)}_{eno,u} \rightarrow S^{(3)}_{eno,uu}$: LFSR unrolling for SNOW 2.0 reduces the area efficiency for SOSEMANUK and improves the same for SNOW 2.0.

$S^{(3)}_{eno,u} \rightarrow S^{(3)}_{eo,u}$: Removing SNOW 2.0 mode decreases the area, and thus, improves the area efficiency further.

$S^{(3)}_{eo,u} \rightarrow S^{(2)}_{eo,u}$: Moving back to the 2-stage pipeline increases the area efficiency for Serpent. Due to the drop of clock frequency and increase of area, the area efficiency for SOSEMANUK is compromised.

$S^{(2)}_{eo,u} \rightarrow S^{(2)}_{eno,u}$: The area efficiency drops further when SNOW 2.0 mode is included in the design.

6.2 Initialization Latency

The initialization latency of the different algorithms for different design points are shown in Table 5. For Serpent, all the design points require 99 cycles for initialization. The initialization involves 33 rounds of key scheduling. For each round, there are two instructions for computing the recurrence followed by a 1-cycle S-box computation.

For SNOW 2.0, the initialization requires 32 initial clocking of the LFSR, which are accomplished in 32 cycles.

For SOSEMANUK, the truncated key schedule requires 25 rounds, with each round consuming 3 cycles similar to Serpent. This is followed by the encryption of IV with Serpent24. This operation requires 1 cycle for initialization of the permutation network indices and 1 additional final cycle, which accesses the round key twice. In between, the 24 rounds require 2 and 1 cycle for 3-stage and 2-stage design variants respectively. The loading of the LFSR, R1 and R2 needs altogether 3 cycles.

In terms of overall performance, the two best design points are $S^{(2)}_{eno,u}$ and $S^{(3)}_{eno,u}$. Whereas $S^{(3)}_{eno,u}$ gives better performance for SNOW 2.0 and SOSEMANUK, $S^{(2)}_{eno,u}$ provides better performance for Serpent.

Table 5: Initialization Latency of Different Algorithms

Design	Core Frequency (MHz)	Initialization Latency (Cycles)		
		Serpent	SNOW 2.0	SOSEMANUK
$S^{(3)}_{e,basic}$	1600	33×3 = 99	-	-
$S^{(3)}_{e}$	1500	33×3 = 99	-	-
$S^{(2)}_{e}$	1060	33×3 = 99	-	-
$S^{(2)}_{en}$	1040	33×3 = 99	32	-
$S^{(2)}_{n}$	1950	-	32	-
$S^{(2)}_{eno}$	1010	33×3 = 99	32	25×3 + 1 + 24×1 + 1 + 3 = 104
$S^{(3)}_{eno}$	1280	33×3 = 99	32	25×3 + 1 + 24×2 + 1 + 3 = 128
$S^{(3)}_{eno,u}$	1060	33×3 = 99	32	25×3 + 1 + 24×2 + 1 + 3 = 128
$S^{(3)}_{eno,uu}$	1060	33×3 = 99	32	25×3 + 1 + 24×2 + 1 + 3 = 128
$S^{(3)}_{eo,u}$	1040	33×3 = 99	-	25×3 + 1 + 24×2 + 1 + 3 = 128
$S^{(2)}_{eo,u}$	1000	33×3 = 99	-	25×3 + 1 + 24×1 + 1 + 3 = 104
$S^{(2)}_{eno,u}$, $S^{'(2)}_{eno,u}$	1000	33×3 = 99	32	25×3 + 1 + 24×1 + 1 + 3 = 104

6.3 Benchmarking with Other Implementations

Before our current work, hardware performance of SOSEMANUK has been discussed only in [16]. According to [16], the maximum clock frequency achieved in 0.13μm Standard Cell CMOS technology was 188.3 MHz, leading to a throughput of 6.026 Gbps with a power of 14702 μW and an energy-efficiency 2.44 pJ/bit. Total area was 18.819 kGates and throughput/area was 61.77 kbps/μm^2. From the throughput figures, it can be observed that [16] generated 32 bits per cycle. Though it is hard to compare across different technology nodes, it can be safely assumed that from 0.13μm to 65nm allows for 2 times improvement (due to Moore's Law [26]) in maximum achievable clock frequency. Our throughput in Gbps is found to be 10 times faster than theirs. Out of this gain, a factor of 2 is due to technology scaling, another factor of 2 is contributed by the LFSR unrolling and the remaining factor of 2.5 is achieved by our microarchitectural optimizations. However, our area is increased by only 5 times, thereby reflecting better area efficiency.

The best hardware for SNOW 2.0 is due to [13]. They implement it on XC4VLX15 series of Xilinx ISE 6.3.03i FPGA and report a throughput of 8.076 Gbps at a clock speed of 252.4 MHz. Throughput per slice was 3.42 Mbps/slice. The throughput in terms of bits per cycle of [13] is same as the proposed design. Absence of standard cell synthesis results prevented us from further benchmarking the performance of SNOW 2.0 algorithm.

The available published hardware implementation of Serpent [23] compared hardware performances of Serpent and Rijndael AES at 0.6μm 3LM technology (AMS CUA). The comparison is as follows.

	Rijndael	Serpent
Actual chip area (mm^2)	49.0	49.0
Throughput in ECB mode (Gbps)	2.26	1.96
Clock frequency (MHz)	88.5	122.9

Compared to the 0.6μm technology, 65nm is 6 technology generations ahead indicating a potential speed-up of $2^{\frac{6}{2}}$, i.e., 8 times [26]. Even our multi-mode design point $S^{(2)}_{eno,u}$ achieves a frequency of 1000 MHz, which is 8.13× more than that reported in [23]. In terms of bits per cycle, [23] achieves 4× more throughput than ours due to their 4× replication of Serpent units.

Though AES and SOSEMANUK are structurally different, it is interesting to note that the highest throughput obtained in our SOSEMANUK implementation outperforms state-of-the-art AES (both software and hardware) implementations [23, 33, 19, 7, 22].

Table 6: Throughputs reported in [27]

Machine	SOSEMANUK		SNOW 2.0	
	cycles/word	Gbps	cycles/word	Gbps
Intel Pentium M (1.7 GHz)	4.68	11.62	4.75	11.45
Intel Pentium 4 (2.8 GHz)	5.81	15.42	5.01	17.88
AMD Athlon 64 X2 4200+ (2.2 GHz)	4.07	17.30	4.83	14.58

Now let us turn to software performance. According to [27], the throughputs of SOSEMANUK and SNOW 2.0 are as given in Table 6. Our proposed accelerator improves these performances by at least 3.7× and 1.8× for SOSEMANUK and SNOW 2.0 respectively.

To compare the throughput of our SOSEMANUK hardware with that of other eSTREAM finalists, we quote the following state-of-the-art throughput results. For hardware category, we have the following pairs of throughputs (in Gbps), the first of which is in 0.25μ [17] and the second in 0.13 μm [16].

- Grain: 4.475 (Grain-v1) & 14.48 (Grain-128),
- MICKEY: 0.287 & 0.413,
- Trivium: 18.568 & 22.3.

For the software category, we have the following throughput figures (in Gbps).

- Salsa20/12 [18]: 6.4 (0.18μm),
- Rabbit [34]: 25.62 (Xilinx Virtex-5 LXT FPGA),
- HC-128 [4]: 22.88 (65nm).

Even after taking into account the technology scaling, our proposed SOSEMANUK implementation beats all the above results. However, it is not clear how FPGA implementation of Rabbit should be interpreted. For a fair comparison, both Rabbit and SOSEMANUK should be implemented in similar technology and we aim to pursue that in our future work.

7 Conclusion and Future Work

We propose a hardware accelerator for the eSTREAM finalist software stream cipher SOSEMANUK. Since the cipher combines the design principles of the block cipher Serpent and the stream cipher SNOW 2.0, we accommodate these two ciphers also in our design. In terms of performance, our hardware beats all stand-alone hardware implementations of all the three ciphers as well as the existing hardwares for all the other ciphers of eSTREAM portfolio.

Because of the complicated design of SOSEMANUK, the hardware area is not suitable for light-weight applications; however, our design can certainly be used as a flexible hardware accelerator serving the purpose of both block and stream ciphers. Additionally, a flexible instruction-set allows experimenting with further design points exploring security-performance trade-offs.

It can be noted that LFSR unrolling of SNOW 3G resulted in diminishing area-efficiency [31]. In that context, the unrolling results of SOSEMANUK, experimented in this work, is encouraging and we intend to probe further unrolling possibility. The area efficiency for SNOW 2.0 with unrolling option can be explored, too.

References

[1] Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard. National Institute of Standards and Technology Docket No. 960924272-6272-01, RIN 0693-ZA13, January 2, 1997 Available at `http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt`.

[2] H. Ahmadi, T. Eghlidos and S. Khazaei. Improved Guess and Determine Attack on SOSEMANUK. Available at `http://www.ecrypt.eu.org/stream/papersdir/085.pdf`, 2005.

[3] Intel AES Instruction Set. Available at `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set`.

[4] A. Chattopadhyay, A. Khalid, S. Maitra and S. Raizada. Designing high-throughput hardware accelerator for stream cipher HC-128. In IEEE ISCAS, pp. 1448–1451, 2012.

[5] E. Biham, R. J. Anderson and L. R. Knudsen. Serpent: A new block cipher proposal. In FSE, pp. 222–238, 1998.

[6] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin and H. Sibert. SOSEMANUK, a fast software-oriented stream cipher. In CoRR, abs/0810.1858, 2008.

[7] J. W. Bos, D. A. Osvik and D. Stefan Fast Implementations of AES on Various Platforms. Available at `eprint.iacr.org/2009/501.pdf`, 2009.

[8] J. Y. Cho and M. Hermelin. Improved Linear Cryptanalysis of SOSEMANUK. In ICISC, pp. 101–117, 2009.

[9] J. Daemen and V. Rijmen. AES Proposal: Rijndael. Federal Information Processing Standards Publication 197, November 26, 2001. Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[10] P. Ekdahl, T. Johansson. SNOW – a new stream cipher. In Proceedings of First Open NESSIE Workshop, Heverlee, Belgium, 2000.

[11] P. Ekdahl and T. Johansson. A New Version of the Stream Cipher SNOW. In SAC, pp. 47–61, 2002.

[12] eSTREAM: the ECRYPT Stream Cipher Project. Available at `http://www.ecrypt.eu.org/stream`

[13] W. H. Fang, T. Johansson and L. Spaanenburg. SNOW 2.0 IP core for trusted hardware. International Conference on Field Programmable Logic and Applications, pp. 281–286, 2005.

[14] Faraday Memory Compiler. Available at `http://www.faraday-tech.com/html/products/freelibrary/memoryCompiler.html`.

[15] X. Feng, J. Liu, Z. Zhou, C. Wu and D. Feng. A Byte-Based Guess and Determine Attack on SOSEMANUK. In ASIACRYPT, LNCS vol. 6477, pp. 146–157, 2010.

[16] T. Good and M. Benaissa. Hardware results for selected stream cipher candidates. In State of the Art of Stream Ciphers (SASC), pp. 191–204, 2007.

[17] F. K. Gürkaynak, P. Luethi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber and W. Fichtner. Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/015, 2006.

[18] L. Henzen, F. Carbognani, N. Felber and W. Fichtner. VLSI hardware evaluation of the stream ciphers Salsa20 and ChaCha, and the compression function Rumba. In IEEE International Conference on Signals, Circuits and Systems, pp. 1–5, 2008.

[19] A. Hodjat, D. D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede. A 3.84 Gbits/s AES crypto coprocessor with modes of operation in a 0.18-μm CMOS technology. In Proceedings of the 15th ACM Great Lakes symposium on VLSI, pp. 60–63, 2005.

[20] P. Kitsos, G. Kostopoulos, N. Sklavos and O. Koufopavlou. Hardware Implementation of the RC4 stream Cipher. In Proc. of 46th IEEE Midwest Symposium on Circuits & Systems, Cairo, Egypt, pp. 1363–1366, 2003.

[21] J. K. Lee, D. H. Lee, S. Park. Cryptanalysis of sosemanuk and SNOW 2.0 using linear masks. In ASIACRYPT, LNCS vol. 5350, pp. 524–538, 2008.

[22] Ruby B. Lee and Yu-Yuan Chen. Processor accelerator for AES. In Proceedings of the 8th IEEE Symposium on Application Specific Processors (SASP), pp. 16–21, doi: 10.1109/SASP.2010.5521153, 2010.

[23] A. K. Lutz, J. Treichler, F. K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker and W. Fichtner. 2Gbit/s Hardware Realizations of RIJNDAEL and SERPENT: A Comparative Analysis. In CHES, LNCS vol. 2523, pp. 144–158, 2003.

[24] Z. Ma and D. Gu. Improved Differential Fault Analysis of SOSEMANUK. In 8th IEEE International Conference on Computational Intelligence and Security, pp. 487–491, 2012.

[25] D. P. Matthews Jr. Methods and apparatus for accelerating ARC4 processing. US Patent Number 7403615, Morgan Hill, CA, July, 2008. Available at `http://www.freepatentsonline.com/7403615.html`.

[26] G. E. Moore. Cramming more components onto integrated circuits. In Electronics, vol. 38, no. 8, April 19, 1965, pp. 114–117.

[27] Performance Figures. The eSTREAM Project - eSTREAM Phase 3. Available at `http://www.ecrypt.eu.org/stream/phase3perf.html`.

[28] Y. E. Salehani, A. Kircanski and A. Youssef. Differential Fault Analysis of SOSEMANUK. In AFRICACRYPT, LNCS vol. 6737, pp. 316–331, 2011.

[29] P. Schaumont and I. Verbauwhede. Hardware/software codesign for stream ciphers. In State of the Art of Stream Ciphers (SASC), 2007. Available at `http://www.ecrypt.eu.org/stream/papersdir/2007/016.pdf`.

[30] S. Sen Gupta, A. Chattopadhyay and A. Khalid. HiPAcc-LTE: An Integrated High Performance Accelerator for 3GPP LTE Stream Ciphers. In INDOCRYPT, pp. 196–215, 2011.

[31] S. Sen Gupta, A. Chattopadhyay and A. Khalid. Designing Integrated Accelerator for Stream Ciphers with Structural Similarities. In Cryptography and Communications 5(1), pp. 19–47, 2013.

[32] S. Sen Gupta, A. Chattopadhyay, K. Sinha, S. Maitra and B. P. Sinha. High Performance Hardware Implementation for RC4 Stream Cipher. In IEEE Transactions on Computers, doi: 10.1109/TC.2012.19, 2012.

[33] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater and J.-D. Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In CHES, pp. 334–350, 2003.

[34] D. Stefan. Hardware Framework for the Rabbit Stream Cipher. In Inscrypt, pp. 230–247, 2009.

[35] Synopsys Design Compiler. Available at `http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx`.

[36] Synopsys PrimeTime. Available at `http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx`.

[37] Synopsys Processor Designer. Available at `http://www.synopsys.com/Systems/BlockDesign/processorDev`.

[38] T. H. Tran, L. Lanante, Y. Nagao, M. Kurosaki and H. Ochi. Hardware Implementation of High Throughput RC4 Algorithm. In IEEE ISCAS, pp. 77–80, 2012.

[39] Y. Tsunoo, T. Saito, M. Shigeri, T. Suzaki, H. Ahmadi, T. Eghlidos and S. Khazaei. Evaluation of SOSEMANUK With Regard to Guess-and-Determine Attacks. Available at `http://www.ecrypt.eu.org/stream/papersdir/2006/009.pdf`, 2006.

[40] J. Yan and H. M. Heys. Hardware Implementation of the Salsa20 and Phelix Stream Ciphers. In IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), pp. 1125–1128, 2007.

[41] X. Zhang and K. Parhi. An Efficient 21.56 Gbps AES Implementation on FPGA. In Thirty-Eighth Asilomar Conference on Signals, Systems and Computers. pp. 465–470. 2004.

A Serpent S-box, Linear Transformation and Key Scheduling

In this appendix, we describe some relevant parts of Serpent specification.

A.1 Bitsliced Implementation of Serpent S-boxes

In [5], the eight Serpent S-boxes act on 4-bit words and are defined as permutations of $\mathbb{Z}_{16}$, as follows.

$$
\begin{array}{rcl}
S_0 & : & 3, 8, 15, 1, 10, 6, 5, 11, 14, 13, 4, 2, 7, 0, 9, 12 \\
S_1 & : & 15, 12, 2, 7, 9, 0, 5, 10, 1, 11, 14, 8, 6, 13, 3, 4 \\
S_2 & : & 8, 6, 7, 9, 3, 12, 10, 15, 13, 1, 14, 4, 0, 11, 5, 2 \\
S_3 & : & 0, 15, 11, 8, 12, 9, 6, 3, 13, 1, 2, 4, 10, 7, 5, 14 \\
S_4 & : & 1, 15, 8, 3, 12, 0, 11, 6, 2, 5, 4, 10, 9, 14, 7, 13 \\
S_5 & : & 15, 5, 2, 11, 4, 10, 9, 12, 0, 3, 14, 8, 13, 6, 7, 1 \\
S_6 & : & 7, 2, 12, 5, 8, 4, 6, 11, 14, 9, 1, 15, 13, 3, 10, 0 \\
S_7 & : & 1, 13, 15, 0, 14, 8, 2, 11, 7, 4, 12, 10, 9, 3, 5, 6
\end{array}
$$

In each round, 32 copies of one S-box is applied in parallel to map 128-bit input to 128-bit output. In bitslice mode, each S-box takes 4 32-bit words as inputs and produces 4 32-bit words as outputs. The bitsliced implementation is described below. The notations ^ , &, | and ~ mean bitwise XOR, AND, OR and NOT operations respectively. The notation $a\langle op\rangle = b$ means $a = a\langle op\rangle b$. In each case, r0, r1, r2, r3 act as the input words and r3 act as an auxilliary variable. The output indices are given in the second column of Table 4.1.

```
S-box S0(r0, r1, r2, r3, r4) {
        r3 ^= r0; r4 = r1;
        r1 &= r3; r4 ^= r2;
        r1 ^= r0; r0 |= r3;
        r0 ^= r4; r4 ^= r3;
        r3 ^= r2; r2 |= r1;
        r2 ^= r4; r4 = ~r4;
        r4 |= r1; r1 ^= r3;
        r1 ^= r4; r3 |= r0;
        r1 ^= r3; r4 ^= r3;
}

S-box S1(r0, r1, r2, r3, r4) {
        r0 = ~r0; r2 = ~r2;
        r4 = r0; r0 &= r1;
        r2 ^= r0; r0 |= r3;
        r3 ^= r2; r1 ^= r0;
        r0 ^= r4; r4 |= r1;
        r1 ^= r3; r2 |= r0;
        r2 &= r4; r0 ^= r1;
        r1 &= r2;
        r1 ^= r0; r0 &= r2;
        r0 ^= r4;
}

S-box S2(r0, r1, r2, r3, r4) {
        r4 = r0; r0 &= r2;
        r0 ^= r3; r2 ^= r1;
        r2 ^= r0; r3 |= r4;
        r3 ^= r1; r4 ^= r2;
        r1 = r3; r3 |= r4;
        r3 ^= r0; r0 &= r1;
        r4 ^= r0; r1 ^= r3;
        r1 ^= r4; r4 = ~r4;
}

S-box S3(r0, r1, r2, r3, r4) {
        r4 = r0; r0 |= r3;
        r3 ^= r1; r1 &= r4;
        r4 ^= r2; r2 ^= r3;
        r3 &= r0; r4 |= r1;
        r3 ^= r4; r0 ^= r1;
        r4 &= r0; r1 ^= r3;
        r4 ^= r2; r1 |= r0;
```

```
        r1 ^= r2; r0 ^= r3;
        r2 = r1; r1 |= r3;
        r1 ^= r0;
}

S-box S4(r0, r1, r2, r3, r4) {
        r1 ^= r3; r3 = ~r3;
        r2 ^= r3; r3 ^= r0;
        r4 = r1; r1 &= r3;
        r1 ^= r2; r4 ^= r3;
        r0 ^= r4; r2 &= r4;
        r2 ^= r0; r0 &= r1;
        r3 ^= r0; r4 |= r1;
        r4 ^= r0; r0 |= r3;
        r0 ^= r2; r2 &= r3;
        r0 = ~r0; r4 ^= r2;
}

S-box S5(r0, r1, r2, r3, r4) {
        r0 ^= r1; r1 ^= r3;
        r3 = ~r3; r4 = r1;
        r1 &= r0; r2 ^= r3;
        r1 ^= r2; r2 |= r4;
        r4 ^= r3; r3 &= r1;
        r3 ^= r0; r4 ^= r1;
        r4 ^= r2; r2 ^= r0;
        r0 &= r3; r2 = ~r2;
        r0 ^= r4; r4 |= r3;
        r2 ^= r4;
}

S-box S6(r0, r1, r2, r3, r4) {
        r2 = ~r2; r4 = r3;
        r3 &= r0; r0 ^= r4;
        r3 ^= r2; r2 |= r4;
        r1 ^= r3; r2 ^= r0;
        r0 |= r1; r2 ^= r1;
        r4 ^= r0; r0 |= r3;
        r0 ^= r2; r4 ^= r3;
        r4 ^= r0; r3 = ~r3;
        r2 &= r4;
        r2 ^= r3;
}

S-box S7(r0, r1, r2, r3, r4) {
        r4 = r1; r1 |= r2;
        r1 ^= r3; r4 ^= r2;
        r2 ^= r1; r3 |= r4;
        r3 &= r0; r4 ^= r2;
        r3 ^= r1; r1 |= r4;
        r1 ^= r0; r0 |= r4;
        r0 ^= r2; r1 ^= r4;
        r2 ^= r1; r1 &= r0;
        r1 ^= r4; r2 = ~r2;
        r2 |= r0;
        r4 ^= r2;
}
```

A.2 Bitsliced Implementation of Serpent Linear Transformation

Here $ROTL(a, b)$ means rotate the 32-bit word a by b positions and $T32(a)$ means truncate a to its lower-order 32 bits.

```
LT(x0, x1, x2, x3) {
        x0 = ROTL(x0, 13);
        x2 = ROTL(x2, 3);
        x1 = x1 ^ x0 ^ x2;
        x3 = x3 ^ x2 ^ T32(x0 << 3);
        x1 = ROTL(x1, 1);
        x3 = ROTL(x3, 7);
```

```
        x0 = x0 ^ x1 ^ x3;
        x2 = x2 ^ x3 ^ T32(x1 << 7);
        x0 = ROTL(x0, 5);
        x2 = ROTL(x2, 22);
}
```

A.3 Serpent Subkey Generation

The S-boxes are used to transform the prekeys $w_0, \ldots, w_7$ into words k_i of the round keys as follows.

$$\begin{aligned}
\{k_0, k_1, k_2, k_3\} &= S_3(w_0, w_1, w_2, w_3) \\
\{k_4, k_5, k_6, k_7\} &= S_2(w_4, w_5, w_6, w_7) \\
\{k_8, k_9, k_{10}, k_{11}\} &= S_1(w_8, w_9, w_{10}, w_{11}) \\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &= S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &= S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\ldots \\
\{k_{124}, k_{125}, k_{126}, k_{127}\} &= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &= S_3(w_{128}, w_{129}, w_{130}, w_{131}).
\end{aligned}$$

Now, the i-th subkey is formed as

$$K_i = \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}.$$

The above assumes bitsliced implementation of the S-boxes. Otherwise IP needs to be applied to the round keys to place the key bits in proper position.

B SNOW 2.0 Key Initialization

According to the SNOW 2.0 specification [11], the cipher supports a secret key K of either 128 or 256 bits and a 128-bit initialization vector $IV = (IV_3, IV_2, IV_1, IV_0)$. The 128-bit key is denoted by $(k_3, \ldots, k_0)$ and the 256-bit key is denoted by $(k_7, \ldots, k_0)$. For the 128-bit case, the LFSR is loaded as follows.

$$\begin{aligned}
&s_{15} = k_3 \oplus IV_0, \; s_{14} = k_2, && s_{13} = k_1, && s_{12} = k_0 \oplus IV_1, \\
&s_{11} = k_3 \oplus \mathbf{1}, && s_{10} = k_2 \oplus \mathbf{1} \oplus IV_2, && s_9 = k_1 \oplus \mathbf{1} \oplus IV_3, \; s_8 = k_0 \oplus \mathbf{1}, \\
&s_7 = k_3, && s_6 = k_2, && s_5 = k_1, && s_4 = k_0, \\
&s_3 = k_3 \oplus \mathbf{1}, && s_2 = k_2 \oplus \mathbf{1}, && s_1 = k_1 \oplus \mathbf{1}, && s_0 = k_0 \oplus \mathbf{1}.
\end{aligned}$$

For the 256-bit case, it is loaded as

$$\begin{aligned}
&s_{15} = k_7 \oplus IV_0, \; s_{14} = k_6, && s_{13} = k_5, && s_{12} = k_4 \oplus IV_1, \\
&s_{11} = k_3, && s_{10} = k_2 \oplus IV_2, && s_9 = k_1 \oplus IV_3, \; s_8 = k_0,
\end{aligned}$$

and $s_i = k_1 \oplus \mathbf{1}$ for $i = 0, \ldots, 7$. Next, the LFSR is clocked 32 times without producing any output and the new element to be inserted is given by

$$s_{t+16} = \alpha^{-1} S_{t+11} \oplus s_{t+2} \oplus \alpha s_t \oplus F_t.$$