On the Parallelization of the MICKEY-128 2.0 Stream Cipher

Deian Stefan and Christopher Mitchell

S*ProCom[2] // Dept. of Electrical Engineering,
The Cooper Union
New York, NY 10003
{stefan,mitche2}@cooper.edu

Abstract. The increasingly widespread use of electronic devices necessitates efficient stream ciphers providing high-throughput encryption with minimal resource usage. MICKEY-128 2.0 is a recent hardware-oriented synchronous stream cypher with a 128-bit key, proposed to the ECRYPT Stream Cipher Project. Using a novel mathematical interpretation of the algorithm, we present a method of parallelizing the stream cipher to produce an n-bit keystream output. We demonstrate a high-throughput (560 Mbps), area-efficient (392 slices) two-way parallelized implementation on the Xilinx Virtex-II Pro FPGA.

Key words: Cryptography, FPGA, MICKEY-128, Matrix analysis.

1 Introduction

As communications security becomes a higher priority across a range of applications, demand is growing for encryption implementable on hardware of widely divergent capabilities. Stream ciphers are cryptographic algorithms that operate on a continuous *stream* of plaintext data (usually one bit a time) to produce the same amount of ciphertext [13]. They are designed to fulfill the requirements of high-throughput, low-resource usage, and high security [11]. Although they are not as popular as *block ciphers* (DES and AES), stream ciphers are used in such varied applications as the Bluetooth protocol[1], the Group Special Mobile (GSM) cellular phone standard[2], and the Secure Socket Layer (SSL)[3]. MICKEY-128 (Mutual Irregular Clocking KEYstream generator) version 2.0 is a recent stream cipher under consideration as a final candidate by the European Network of Excellence for Cryptography (*eCRYPT*) organization under the *eSTREAM*[4] project.

We first introduce the proposed MICKEY-128 2.0 stream cipher, then present a matrix–algebra interpretation of the cryptographic algorithm that allows for

[1] E0 stream cipher.
[2] A5/1 stream cipher.
[3] RC4 stream cipher.
[4] http://www.ecrypt.eu.org/stream/

the generation of n bits at a time; this improves on previous implementations that only generate a single bit a time. Finally, we present two hardware implementations on *Field-Programmable Gate Arrays* (FPGAs).

2 MICKEY-128 2.0 Stream Cipher

MICKEY-128 2.0 is a hardware-oriented *binary additive stream cipher* designed to have a long period (2^{64} bit keystream sequence) and high security while maintaining low complexity [2]. Figure 1 shows the overall design of a binary additive stream cipher; the keystream generator is initialized by a secret key (known only to the communicating parties) and an initial variable (IV) (which can be public) from which it generates a keystream z_k which is simply added (XOR) to the plaintext m_k to form the ciphertext c_k. The receiving party *symmetrically* generates the same keystream z_k and adds it to the received ciphertext c_k to get the original plaintext (since $c_k \oplus z_k = [z_k \oplus m_k] \oplus z_k = m_k$).

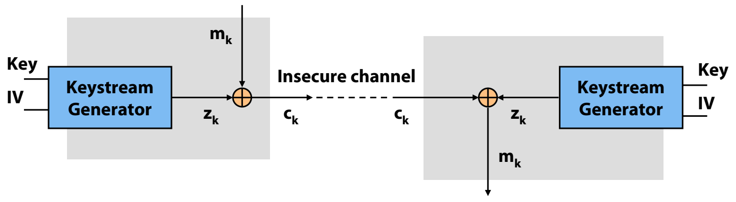

Fig. 1. Symmetric Encoding

The design of MICKEY 128-2.0 is an improvement on the MICKEY 2.0 [1], with a 128-bit key and longer keystream period. The increase in key length from 80- to 128-bit addresses *brute-force* attacks; if an attacker should try an exhaustive search for the correct key she would now have $\approx 28 \times 10^{13}$ additional possible keys to search through. The increase of the state size provides strong protection again Biryukov-Shamir Time-Memory-Data attacks [1].

Note: We shall refer to MICKEY-128 2.0 simply as MICKEY from this point.

2.1 Clocking the Keystream Generator

The MICKEY architecture is based on two 160-bit registers R and S; the state of *both* the R and S registers controls the clocking of each register in a method called *mutual irregular clocking*. Following the notation of [2], the pseudocode defining the clocking of the keystream generator is shown below:

```
CLOCK-KG($mix, I$)
1  $c_{\mathrm{r}} \leftarrow S[54] \oplus R[106]$
```

```
2  $c_S \leftarrow S[106] \oplus R[53]$
3  $i_r \leftarrow I \oplus (S[80] \cdot mix)$
4  Clock-R($i_r, c_r$)
5  Clock-S($I, c_S$)
```

where $\oplus$ and $\cdot$ are the bitwise XOR and AND operations, respectively, and $R[i]$ and $S[i]$ are the ith bits of registers R and S, respectively.

The mix and input bit I are used when initializing the generator (using bits from the key and IV), as explained in [2]. After initialization, however, both bits are set to zero and the keystream output (at each clocking of CLOCK-KG) is the combination of the first bits of R and S:

$$z_k = R[0] \oplus S[0] \tag{1}$$

The use of bits from both registers to generate the keystream output is to prevent *algebraic* attacks which take advantage of the correlation between the output and linearity of one the registers (specifically, R); see Section 7 of [1] for more details on the design principles of MICKEY.

2.2 Clocking the *R* Register

The "engine" behind the keystream generator is the 160-bit register R, where the leftmost bit is labeled R[159] and the rightmost is R[0]. Babbage and Dodd define register R as the *linear* register; when the control bit $c_r = 0$ register R is a Galois–style Linear Feedback Shift Register (LFSR) whose characteristic polynomial $C_R(x)$ is determined by the tap positions RTAPS, as explained in [2]. Clocking a Galois–LFSR is performed by shifting the register and adding the feedback bit f to the taps; Figure 2 illustrates this idea. After each clocking the *updated* register is labeled R'. When the control bit $c_r = 1$, in addition to shifting and adding the feedback bit, the previous (buffered) state bits of R are added to R'. The pseudocode below defines the clocking of R:

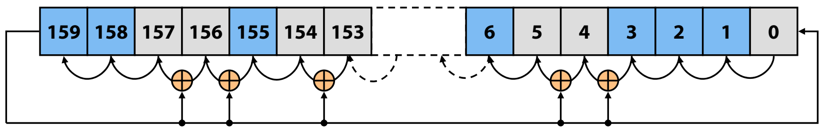

Fig. 2. Galois LFSR

```
Clock-R($i_r, c_r$)
1  $R' \leftarrow R \ll 1$  /*shift left by 1*/
2  $f \leftarrow R[159] \oplus i_r$  /*feedback bit*/
3  if $f = 1$
```

```
4      then $R' \leftarrow R' \oplus T$
5  if $c_{\mathrm{R}} = 1$
6      then $R' \leftarrow R' \oplus R$
7  $R \leftarrow R'$
```

where T is a binary 160-bit vector whose bits are defined as:

$$T[i] = \begin{cases} 1 \text{ if } i \in \text{RTAPS}; \\ 0 \text{ otherwise.} \end{cases} \tag{2}$$

and i_{R} is the input bit used during initialization.

In defining CLOCK-R we have used a higher level of abstraction (similar to the submitted C code implementation by Babbage and Dodd) than the described algorithm presented in [2]. Rather than describing the algorithm as operations on 160 one-bit registers, we are equivalently describing the algorithm in terms of 160-bit vector operations.

2.3 Clocking the S Register

Using the same level of abstraction as CLOCK-R we define the clocking of S, the *nonlinear* register, according to the pseudocode below (note that the pseudocode is equivalent to the clocking of S as originally defined in [2]):

```
CLOCK-S($i_{\mathrm{S}}, c_{\mathrm{S}}$)
 1  $K_0 \leftarrow (0,1,1,\ldots,1,1,0)$  /*160-bit mask*/
 2  $K_1 \leftarrow (1,0,0,\ldots,0,0,0)$
 3  $S_1 \leftarrow S \ll 1$
 4  $S_2 \leftarrow S \gg 1$
 5  $f \leftarrow S[159] \oplus i_{\mathrm{S}}$  /*feedback bit*/
 6  $p \leftarrow S[158]$
 7  $S' \leftarrow (S_1 \oplus ((S \oplus \mathrm{COMP}_0) \cdot (S_2 \oplus \mathrm{COMP}_1))) \cdot K_0$
 8  if $p = 1$
 9      then $S' \leftarrow S' \oplus K_1$
10  if $f = 1$
11      then
12              if $c_{\mathrm{S}} = 1$
13                  then $S' \leftarrow S' \oplus \mathrm{FB}_1$
14                  else  $S' \leftarrow S' \oplus \mathrm{FB}_0$
15  $S \leftarrow S'$
```

To keep all the "instructions" as 160-bit vector operations, we extend the 158-bit COMP_i vectors (constants COMP_i and FB_i are defined in [2]) to 160-bit vectors by padding the beginning and end with 0's. Furthermore, rather than explicitly assigning bits $S'[159] = S[158]$ and $S'[0] = 0$ we use masks K_0 and K_1 (lines 7-9); in doing so we will later be able to express CLOCK-S using simple algebraic methods (see Section 3). Note that unlike clocking register R, when the control bit $c_{\mathrm{S}} = 0$ the clocking of S does not reduce to that of a simple Galois–LFSR.

Although R is the "engine" of the keystream generator, simply using the output of a *linear* register as the keystream will allow an adversary to easily determine future stream values [11]. The use of the S register as part of the keystream output and *mutual* clocking of the two registers as specified by Babagge and Dodd avoids these *distinguishing* attacks [1], and therefore strengthens the security of MICKEY.

3 Parallelizing the Algorithm

The cryptographic algorithm is parallelized $n\times$ by *looking-ahead* for $n-1$ future feedback, control, and keystream bits $\{f^{(i)}, c^{(i)}, z_{k+i} : 1 \leq i < n\}$ and then clocking the R and S registers n times ahead. For example, if we wish to obtain four output bits $z_{k+i} (0 \leq i \leq 3)$ at once, at current state k we need to look-ahead for three control, feedback, and output bits which are then used to clock the two registers four times ahead.

In [1, 2], Babbage and Dodd suggest a method of generating multiple keystream bits using lookup tables (LUTs). We considered a table–based method, however, it proved to be less efficient than the method we present next. See the Appendix for details on the table-based optimization.

3.1 Calculating the Critical Look-ahead Bits

At each clocking of the keystream generator we calculate the upcoming control, feedback, and keystream output bits and then clock registers R and S. Examining the algorithm, however, we observe that future control and feedback bits can be determined without having to fully clock the two registers again; the control, feedback, and keystream bits only depend on the clocking of a small number of bits from R and S. We refer to these bits as the *critical* bits of R and S.

From the CLOCK-KG pseudocode (see Section 2.1) the upcoming control bits (c'_{r} and c'_{S}) and upcoming keystream output (z_{k+1}) depend on the following look-ahead critical bits: $R'[0], R'[53], R'[106], S'[0], S'[54]$, and $S'[106]$. To correctly parallelize the algorithm we need to clock the two registers twice ahead; to do so we also need to determine the upcoming feedback bit (f') for both registers.

Determining the look-ahead critical bits of R (considering the values of $\mathrm{FB}_i[\cdot]$ and whether $R[\cdot]$ is a tap) is simply accomplished by clocking each bit ($R[\cdot]$) individually:

$$\begin{aligned} R'[159] &= R[158] \oplus (R[159] \cdot c_{\mathrm{r}}) = f' \\ R'[106] &= (R[105] \oplus f) \oplus (R[106] \cdot c_{\mathrm{r}}) \\ R'[53] &= (R[52] \oplus f) \oplus (R[53] \cdot c_{\mathrm{r}}) \\ R'[0] &= f \oplus (R[0] \cdot c_{\mathrm{r}}) \end{aligned} \tag{3}$$

and similarly for S (considering the values of $\mathrm{COMP}_i[\cdot]$),

$$\begin{aligned} S'[159] &= S[158] \oplus (f \cdot \overline{c_\mathrm{S}}) = f' \\ S'[158] &= S[157] \oplus (S[158] \cdot (S[159] \oplus 1)) = p' \\ S'[106] &= S[105] \oplus (S[106] \cdot S[107]) \oplus (f \cdot c_\mathrm{S}) \\ S'[54] &= S[53] \oplus (S[54] \cdot S[55]) \oplus (f \cdot c_\mathrm{S}) \\ S'[0] &= f \end{aligned} \tag{4}$$

where the bar operator on c_S is defined as the NOT operation

$$\overline{c_\mathrm{S}} = 1 - c_\mathrm{S} \ , \tag{5}$$

so that $\overline{0} = 1$ and $\overline{1} = 0$.

Using the critical bits of (3) and (4) we can determine the upcoming keystream output

$$z_{k+1} = R'[0] \oplus S'[0] \ , \tag{6}$$

and calculate the look-ahead control bits

$$c'_\mathrm{T} = S'[54] \oplus R'[106] \text{ and } c'_\mathrm{S} = S'[106] \oplus R'[53] \ . \tag{7}$$

These bits (and the feedback bits) are then used to clock registers R and S ahead by two. Note that in the above equations we assume that the keystream generator has already been initialized, so $mix = 0$ and $I = 0$. To consider the initialization, (3) and (4) need to be slightly altered.

It is important to see that if we wish to clock by $n > 2$ we need to also determine $R''[\cdot], R'''[\cdot], \ldots, R^{(n-1)}[\cdot]$ and $S''[\cdot], S'''[\cdot], \ldots, S^{(n-1)}[\cdot]$, which depend on an increasing number of individually–clocked critical bits. This increases the overall complexity of the look-ahead architecture.

For example, if we wish to clock ahead by three, to calculate $R''[159]$ we need $f' = R'[159]$ and $R'[158]$. The feedback bit f' was already calculated, however $R'[158]$ was not; therefore, we need to clock $R[158]$ which depends on an additional critical bit $R[157]$.

As n increases, the number of critical bits approaches the size of the registers and the complexity of the look-ahead architecture becomes impractical.

3.2 Clocking *R* Ahead *n* Times

To distinguish between matrices and vectors, in the following two sections we follow a different notation: registers R and S and constant T are represented by row vectors $\mathbf{r}, \mathbf{s}$, and $\mathbf{t}$.

Thus far we have represented the algorithm for clocking register R (vector $\mathbf{r}$) using pseudocode CLOCK-R . The short algorithm, which simply performs linear operations on $\mathbf{r}$ can instead be expressed more compactly using simple *matrix algebra* according to:

$$h(\mathbf{r}) = \mathbf{r}' = \mathbf{A}\mathbf{r} + (f\mathbf{I})\mathbf{t} + (c_\mathrm{T}\mathbf{I})\mathbf{r} = [\mathbf{A} + (c_\mathrm{T}\mathbf{I})]\mathbf{r} + (f\mathbf{I})\mathbf{t} \ , \tag{8}$$

where $\mathbf{I}$ is a 160×160 identity matrix and $\mathbf{A}$ is defined as:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{9}$$

an operator performing a left shift by 1 (line 1 of CLOCK-R is equivalent to $\mathbf{Ar}$). Note that all the operations and elements of the vectors and matrices in (8) are modulo 2, so the $+$ operator is simply a bitwise XOR on the binary vectors.

Since we are interested in clocking forward, clocking by n can be accomplished by clocking (8) n times

$$\mathbf{r}^{(n)} = h(\mathbf{r}^{(n-1)}) = h(h(\cdots h(h(\mathbf{r}))\cdots)) \tag{10}$$

while keeping track of all the feedback and control bits $\{f^{(i)}, c_{\mathrm{r}}^{(i)} : (0 \leq i < n)\}$.

For example, according to the above equation, clocking twice is equivalent to clocking (8) again, such that

$$\mathbf{r}'' = \varphi(\mathbf{r}) = h(h(\mathbf{r})) = [\mathbf{A} + (c'_{\mathrm{r}}\mathbf{I})]([\mathbf{A} + (c_{\mathrm{r}}\mathbf{I})]\mathbf{r} + (f\mathbf{I})\mathbf{t}) + (f'\mathbf{I})\mathbf{t} \ , \tag{11}$$

where f' and c'_{r} are the upcoming feedback and control bits calculated using the method previously discussed in Section 3.1.

To achieve the desired result (clocking by n), clocking once every "iteration" according to (10) would take $\sim n$ iterations. However, if $n = ab$ the number of iterations is $O(a+b)$ since we already clocked by b to get $\mathbf{r}^b$ and a to get $\mathbf{r}^a = \tau(\mathbf{r})$ we simply have to clock b using the "clock-by-$a\times$" equation $\tau(\mathbf{r}^b) = \mathbf{r}^{ab}$. Furthermore, if $b \mid a$ then it only takes $\sim a$ iterations. More importantly, if $n = 2^k$, it is only necessary to go through $\sim \lg n$ equations to derive the desired result for clocking $\mathbf{r}$.

For example, to clock $\mathbf{r}$ in advance by four, we can directly clock $\mathbf{r}$ four times using (10) or equivalently clock $\mathbf{r}''$ one time using (11):

$$\mathbf{r}^{(4)} = \phi(\mathbf{r}) = h(h(h(h(\mathbf{r})))) = \varphi(\mathbf{r}'') \ , \tag{12}$$

which is

$$\begin{aligned} \phi(\mathbf{r}) &= [\mathbf{A} + (c'''_{\mathrm{r}}\mathbf{I})]([\mathbf{A} + (c''_{\mathrm{r}}\mathbf{I})]\mathbf{r}'' + (f''\mathbf{I})\mathbf{t}) + (f'''\mathbf{I})\mathbf{t} \\ &= [\mathbf{A} + (c'''_{\mathrm{r}}\mathbf{I})]([\mathbf{A} + (c''_{\mathrm{r}}\mathbf{I})]([\mathbf{A} + (c'_{\mathrm{r}}\mathbf{I})]([\mathbf{A} + (c_{\mathrm{r}}\mathbf{I})]\mathbf{r} + \\ &\quad +(f\mathbf{I})\mathbf{t}) + (f'\mathbf{I})\mathbf{t}) + (f''\mathbf{I})\mathbf{t}) + (f'''\mathbf{I})\mathbf{t} \ . \end{aligned} \tag{13}$$

Similarly, clocking ahead by eight is most efficiently accomplished by clocking $\mathbf{r}^{(4)}$ once using $\phi(\mathbf{r})$ of (13):

$$\begin{aligned} \varsigma(\mathbf{r}) = {} & [\mathbf{A} + (c_{\mathrm{r}}^{(7)}\mathbf{I})]([\mathbf{A} + (c_{\mathrm{r}}^{(6)}\mathbf{I})]([\mathbf{A} + (c_{\mathrm{r}}^{(5)}\mathbf{I})]([\mathbf{A} + (c_{\mathrm{r}}^{(4)}\mathbf{I})]\phi(\mathbf{r}) + \\ & +(f^{(4)}\mathbf{I})\mathbf{t}) + (f^{(5)}\mathbf{I})\mathbf{t}) + (f^{(6)}\mathbf{I})\mathbf{t}) + (f^{(7)}\mathbf{I})\mathbf{t} \end{aligned} \tag{14}$$

taking a total of $\sim \ln n = 3$ iterations to derive the final $8\times$ clocking equation.

Note: In implementing the above, (14) needs to be expressed in terms of $\mathbf{r}$ (simply by replacing $\phi(\mathbf{r})$ with the expanded (13)). If clocking by $n > 8$, this is also necessary to keep track of all the look-ahead feedback and control bits.

3.3 Clocking S Ahead n Times

Analogous to representing CLOCK-R with (8), although significantly more complicated, we can represent CLOCK-S with:

$$k(\mathbf{s}) = (\mathbf{K}_0 + p\mathbf{K}_1)(\mathbf{s}.(\mathbf{B}\mathbf{s}) + [\mathbf{A} + \mathbf{C}_0\mathbf{B} + \mathbf{C}_1]\mathbf{s} + \mathbf{I}\mathbf{c}_k) + f c_{\mathrm{S}}\mathbf{I}\mathbf{f}_1 + f\overline{c_{\mathrm{S}}}\mathbf{I}\mathbf{f}_0 \ , \quad (15)$$

where . is element-wise multiplication modulo 2 (implemented using bitwise AND), $\mathbf{f}_i$ are the FB$_i$ vectors and vector $\mathbf{c}_k = \mathbf{c}_0.\mathbf{c}_1$ and $\mathbf{c}_i$ are the COMP$_i$ vectors, from which matrices $\mathbf{C}_i = \mathrm{diag}(\mathbf{c}_i)$. Row vectors $\mathbf{k}_i$ are the the bit-masks K_i defined in CLOCK-S and $\mathbf{K}_i = \mathrm{diag}(\mathbf{k}_i)$. Multiplication by a diagonal matrix is equivalent to element-wise multiplication, so for example

$$\mathbf{C}_1\mathbf{s} = \mathrm{diag}(\mathbf{c}_1)\mathbf{s} \equiv \mathbf{c}_1.\mathbf{s} \ , \quad (16)$$

which can be implemented by a bitwise AND of the vectors $\mathbf{c}_1$ and $\mathbf{s}$. For clarity we express (15) using all the constants of CLOCK-S . However when deriving the equations for clocking by $n > 2$ (e.g $n = 8$) it is important to combine terms (e.g $\mathbf{L} = \mathbf{A} + \mathbf{C}_0\mathbf{B} + \mathbf{C}_1$). Finally, matrix $\mathbf{B}$ is defined as

$$\mathbf{B} = \mathbf{A}^{\mathrm{T}} \ , \quad (17)$$

a right shift by one operator, so that $\mathbf{B}\mathbf{s}$ is equivalent to line 4 of CLOCK-S .

In the same manner that we clocked $\mathbf{r}$ in (8), clocking $\mathbf{s}$ n times is directly accomplished by:

$$\mathbf{s}^{(n)} = k(\mathbf{s}^{(n-1)}) = k(k(\cdots k(k(\mathbf{s}))\cdots)) \ , \quad (18)$$

from which clocking by two is

$$\mathbf{s}'' = \vartheta(\mathbf{s}) = (\mathbf{K}_0 + p'\mathbf{K}_1)(k(\mathbf{s}).(\mathbf{B}k(\mathbf{s})) + \mathbf{L}k(\mathbf{s}) + \mathbf{I}\mathbf{c}_k) + f'c'_{\mathrm{S}}\mathbf{I}\mathbf{f}_1 + f'\overline{c'_{\mathrm{S}}}\mathbf{I}\mathbf{f}_0 \ . \quad (19)$$

Using the same method of clocking $\mathbf{r}$ n times using $\sim \lg n$ relations, we can clock $\mathbf{s}$ four times ahead by clocking $\mathbf{s}''$ with the expanded (in terms of $\mathbf{s}$) $\vartheta(\mathbf{s})$ of (19) and representing this result by $\mathbf{s}^{(4)} = \theta(\mathbf{s})$ we can clock $\mathbf{s}$ ahead eight times by clocking $\mathbf{s}^{(4)}$ once (effectively four times): $\xi(\mathbf{s}) = \theta(\mathbf{s}^{(4)})$. The results for the 4×- and 8×-clocking are not shown as they provide no additional insight into the method of clocking ahead.

4 Implementation and Discussion

The implemented designs were targeted to the Xilinx *Virtex-II Pro* XC2VP30 FPGA hosted on the XUPV2P evaluation board. The code was written in Verilog

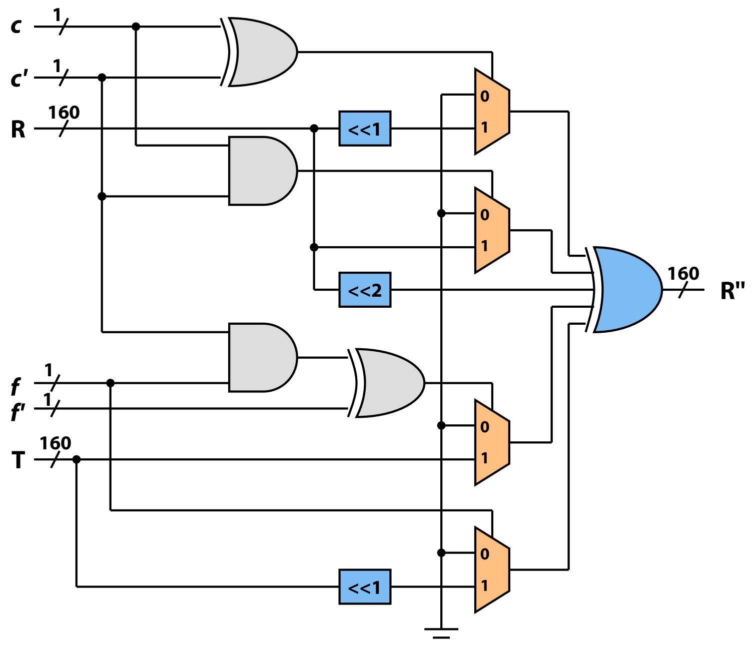

Fig. 3. Clocking R twice ahead.

and synthesized using Synplicity's Synplify Premiere version 9.0.1. The compiled designs were then simulated in Modelsim version 6.2c and the final timing analysis was performed in Modelsim and Premiere for further optimization.

As a proof-of-concept, we implemented two different designs. The first design is a direct implementation of the pseudocode presented in Section 2; the final synthesized design is clocked at 292.9 MHz with a throughput of 292 Mbps using only 1% of the slices available on the FPGA (186 out of 13,696). The second design is an implementation of the stream cipher based on the method of parallelization presented in Section 3. We implemented a 2× clocking of R by factoring (11) and recombining terms to obtain

$$\mathbf{r}'' = \mathbf{AAr} + (c'_{\mathrm{r}} + c_{\mathrm{r}})\mathbf{Ar} + (c'_{\mathrm{r}} c_{\mathrm{r}})\mathbf{Ir} + (c'_{\mathrm{r}} f + f')\mathbf{It} + f\mathbf{At} \ , \qquad (20)$$

which can be easily expressed in pseudocode and logic primitives. Figure 3 shows a possible implementation a synthesis tool might create.

Because the equation for clocking S twice is more complex than that of R, our design is a direct implementation of (19), leaving the simplification to the design tools. The final design is clocked at 280.5 MHz with a throughput of 560 Mbps using 392 slices. Table 1 compares our results with previous implementations of MICKEY and the AES and A5/1 ciphers on similar FPGAs (Virtex and Virtex-II). With the exception of AES, the throughput of the proposed two-way parallelized MICKEY outperforms the other implementations while still maintaining an excellent throughput/area ratio of 1.43.

Table 1. Comparison of different stream cipher implementations

Cipher	Device	Slices	Througput (Gbits/s)	Mbps / Slice
AES[4]	XC2VP50	466	1.30	2.79
A5/1[6]	XC2V250	32	0.19	5.88
MICKEY-128[10]	XCV50	167	0.17	1.02
MICKEY-128 2.0[3]	XC2V6000	190	0.20	1.05
Direct design	XC2VP30	186	0.29	1.58
Parallelized 2×	XC2VP30	392	0.56	1.43

Initialization of the 128-bit key, variable IV, and pre-clocking is performed as explained in [2]. However the number of times R and S are clocked for the second design is half that of the direct implementation (requiring the IV to be even).

5 Conclusion

In this paper we present a method of improving the MICKEY-128 2.0 stream cipher proposed to the eSTREAM project. The eSTREAM project is currently in its final evaluation phase, with MICKEY-128 2.0 being one of the final candidates to be widely adopted.

We implemented a direct (1-bit keystream output) and a two-way parallelized (2-bit keystream output) design of MICKEY-128 2.0 on a Xilinx Virtex-II Pro FPGA to demonstrate the feasibility of maintaining the low-resource, high-throughput, and high-security qualities of the stream cipher. In addition, we considered the algorithmic design for generating an arbitrary number of keystream bits simultaneously. Future work includes further optimization and pipelining of the stream cipher as well as implementations considering 4- and, more importantly, 8-bit keystream generation.

Acknowledgments

The authors would like to thank Fred Fontaine for his support in providing the necessary resources and Ishaan L. Dalal, Jared Harwayne-Gidansky and Sara E. Foley for their insightful comments and suggestions.

References

1. Babbage, S., Dodd, M.: The stream cipher MICKEY 2.0. http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf
2. Babbage, S., Dodd, M.: The stream cipher MICKEY-128 2.0. http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey128_p3.pdf

3. Bulens, P., Kalach, K., Standaert, F-X., Quisquater, J.: FPGA Implementations of eSTREAM Phase-2 Focus Candidates with Hardware Profile. http://www.ecrypt.eu.org/stream/papersdir/2007/024.pdf (2007).
4. Denning, D., et al.: An Implementation of a Gigabit Ethernet AES Encryption Engine for Application Processing in SDR. IEEE 60th Vehicular Technology Conf. 2004-Fall, vol. 3, pp. 19631967 (Sep. 2004).
5. Gaj, K., Southern, G., Bachimanschi, R.: Comparison of hardware performance of selected Phase II eSTREAM candidates. http://www.ecrypt.eu.org/stream/papersdir/2007/026.pdf (2007).
6. Galanis, M.D., Kitsos, P., Kostopoulos, G., Sklavos, N., Koufopavlou, O., Goutis, C.E.: Comparison of the Hardware Architectures and FPGA Implementations of Stream Ciphers. Proceedings of the 2004 11th IEEE International Conf. 2004-Fall, pp. 571-574 (Dec. 2004).
7. Good, T., Benaissa, M.: Hardware Results for Selected Stream Cipher Candidates. http://www.ecrypt.eu.org/stream/papersdir/2007/023.pdf (2007).
8. Gurkaynak, F. K., Luethi, P., Bernold, N., Blattman, R., Goode, V., Marghitola, M., Kaeslin, H., Felber, N., Fichtner, W.: Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt. http://www.ecrypt.eu.org/stream/papersdir/2006/015.pdf (2006).
9. Khazaei, S., Samasizadeh, M., Mohajeri, J.: On the Statistically Optimal Divide and Conquer Correlation Attack on the Shrinking Generator. http://mirror.cr.yp.to/eprint.iacr.org/2005/126.pdf (2005).
10. Kitsos, P.: On the Hardware Implementation of the MICKEY-128 Stream Cipher. http://www.ecrypt.eu.org/stream/papersdir/2006/059.pdf
11. Menezes, A., Van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, London (1996).
12. Rogawski, M.: Hardware evaluation of eSTREAM Candidates: Grain, Lex, Mickey128, Salsa20 and Trivium. http://www.ecrypt.eu.org/stream/papersdir/2007/025.pdf (2007).
13. Schneier, S.: Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition. John Wiley & Sons, Inc. (1996).

Appendix: LUT-based Parallelization

In Section 3 we referred to a different method of generating more than one keystream output bit and clocking the two registers based on lookup tables, as suggested by Babbage and Dodd in [1, 2]. We briefly discuss the table-based approach optimization with respect to register R; however, the same method can be applied to register S.

The most direct method of clocking the register R twice is to duplicate the code of CLOCK-R (see Section 2.2) for the next clocking. This technique is essentially the same as *loop-unrolling* in compiler optimizations, where more than one iterations of the loop are *unrolled* into one iteration. However, most operations of CLOCK-R are conditional (depend on c_{r} and f) and to include the next iteration (in order to clock twice), additional conditions (c'_{r} and f') must also be considered. Table 2 summarizes the operations needed to clock R twice based on the current and upcoming control and feedback bits. It is important to note that the implementation would not be a direct LUT based on Table

2, but would instead be composed of a smaller LUT of operations on T (e.g $(T \ll 1) \oplus T$), which will be added to conditional-transformations of R (e.g. $(R \ll 2) \oplus (R \ll 1)$).

From the table, we see that many of the operations can be easily combined resulting in an implementation would be at least as efficient as our presented method. However, for increasing n the table–based method becomes difficult to simplify without software (the table size is $\sim 2^{2n}$), in comparison to a few lines of algebra using the method we presented in Section 3.

Table 2. Necessary Operations for 2× Optimization

c_r	f	c'_r	f'	Expression for R''
0	0	0	0	$R \ll 2$
0	0	0	1	$(R \ll 2)\oplus T$
0	0	1	0	$(R \ll 2)\oplus(R \ll 1)$
0	0	1	1	$(R \ll 2)\oplus(R \ll 1)\oplus T$
0	1	0	0	$(R \ll 2)\oplus(T \ll 1)$
0	1	0	1	$(R \ll 2)\oplus(T \ll 1)\oplus T$
0	1	1	0	$(R \ll 2)\oplus(T \ll 1)\oplus(R \ll 1)\oplus T$
0	1	1	1	$(R \ll 2)\oplus(T \ll 1)\oplus(R \ll 1)$
1	0	0	0	$(R \ll 2)\oplus(R \ll 1)$
1	0	0	1	$(R \ll 2)\oplus(R \ll 1)\oplus T$
1	0	1	0	$R \ll 2$
1	0	1	1	$(R \ll 2)\oplus R\oplus T$
1	1	0	0	$(R \ll 2)\oplus(R \ll 1)\oplus(T \ll 1)$
1	1	0	1	$(R \ll 2)\oplus(T \ll 1)\oplus(R \ll 1)\oplus T$
1	1	1	0	$(R \ll 2)\oplus(T \ll 1)\oplus T$
1	1	1	1	$(R \ll 2)\oplus(T \ll 1)\oplus R$