# STES: A Stream Cipher Based Low Cost Scheme for Securing Stored Data

Debrup Chakraborty[1], Cuauhtemoc Mancillas-López[1], Palash Sarkar[2]

[1] Department of Computer Science, CINVESTAV-IPN,
Av. IPN 2508 San Pedro Zacatenco, Mexico City 07360
Mexico
debrup@cs.cinvestav.mx, mancilla@computacion.cs.cinvestav.mx

[2] Applied Statistics Unit
Indian Statistical Institute
203 B.T. Road, Kolkata 700108
India
palash@isical.ac.in

**Abstract.** The problem of securing data present on USB memories and SD cards has not been adequately addressed in the cryptography literature. While the formal notion of a tweakable enciphering scheme (TES) is well accepted as the proper primitive for secure data storage, the real challenge is to design a low cost TES which can perform at the data rates of the targeted memory devices. In this work, we provide the first answer to this problem. Our solution, called STES, combines a stream cipher with a XOR universal hash function. The security of STES is rigorously analyzed in the usual manner of provable security approach. By carefully defining appropriate variants of the multi-linear hash function and the pseudo-dot product based hash function we obtain controllable trade-offs between area and throughput. We combine the hash function with the recent hardware oriented stream ciphers, namely Mickey, Grain and Trivium. Our implementations are targeted towards two low cost FPGAs – Xilinx Spartan 3 and Lattice ICE40. Simulation results demonstrate that the speed of encryption/decryption matches the data rates of different USB and SD memories. We believe that our work opens up the possibility of actually putting FPGAs within controllers of such memories to perform low-level in-place encryption.
**keywords:** Tweakable enciphering scheme, stream ciphers, disk encryption, USB memory, SD card, FPGA.

## 1 Introduction

Traditionally, cryptography has been mainly used to secure data in transit. In the last decade, however, there has been an increase in interest in securing stored data. This interest is reflected in some recent standardizing efforts [4] and a galaxy of algorithms that have been proposed for securing stored data [17, 26, 27, 32, 37, 41]. A consensus has been reached among researchers that a type of symmetric encryption scheme, called Tweakable Enciphering Scheme (TES) [26], is appropriate for the application of encrypting data stored in storage devices which have a sector-wise organization including hard disks and NAND flash memories.

The specific application which a TES is meant to serve is called low level disk encryption or in-place disk encryption [26]. In this application, it is assumed that the encryption/decryption algorithm resides in the disk controller and it views the storage media as a collection of sectors. The disk controller encrypts the data before writing and decrypts it after reading and before sending it to the operating system. This generic model of disk encryption is independent of other details like operating systems, file system types, etcetera.

Almost all the known constructions of TES use a block cipher as a building block. Some schemes like CMC, EME, EME* use only block ciphers, whereas schemes such as XCB [32], HCTR [41], HCH [17], TET [25], HEH [36, 37] use a block cipher along with a suitable hash function. In terms of efficiency, the current state-of-the-art for the cost of encryption/decryption per block is either 2 block cipher calls [24, 26, 27]; or, 1 block cipher call plus 1 $GF(2^n)$ multiplication, where $n$ is the size of the underlying block cipher [37].

Though there has been an active effort in designing new TES, there are only a few works which report efficient implementations of such schemes. To our knowledge the only works which report implementations of TES in hardware are [15,31]. The designs in [31] are targeted towards the Xilinx Virtex 4 family of FPGAs while the designs in [15] were targeted towards the Virtex 5 family. The throughput reported in [15] is very encouraging as for all reported designs more than 10 Gbits/sec of throughput is obtained. The best design in terms of speed provides a throughput of more than 15 Gbits/sec. These implementations are prototypical studies and they firmly demonstrate that the speed of TES can match the data rates of modern day disc controllers.

We note that the studies are targeted towards high end FPGAs and so it may not be cost efficient for large scale deployment in commercial hard disks. On the other hand, the design philosophies adopted in these works can be easily adopted for design of ASICs. One can then expect the throughput rates to be higher as ASICs are capable of operating at much higher frequencies compared to FPGAs. Use of ASIC on the other hand, will involve a longer design cycle and the loss of reconfigurability.

## 1.1 The Case for Low Cost Solutions

Storage is an integral part of numerous modern devices. For example, non-trivial storage is provided in modern smart phones and cameras. Even personal bio-medical instruments such as meters for measuring serum glucose or blood pressure have facilities for storing past readings. From a user point of view, the data present on the smart phone of a top political or business personality is no less sensitive than the data present on his or her lap top. With devices being increasingly interconnected and also connected to the net, it may be easy for a piece of malware to load itself into a smart phone and transmit sensitive data. Keeping the stored data in an encrypted form provides a baseline protection to such malicious activity.

Most resource constrained devices do not have hard disks but rely on flash memories. NAND type flash memory has a similar organization as hard disks. Consequently, TES algorithms that are applicable for hard disks can also be applied here. But, one needs to keep in mind the constraints in these devices in terms of area and power utilization. A significant increase in the area will lead to an increase in both the cost and the size of the device. Further, a high demand on power will drain out the battery much sooner. Both of these will negatively impact the utility of the device to the user.

A basic constraint for deployment of any cryptographic protection mechanism is that the performance of the system should not degrade. In the context of stored data, this means that the speed of encryption and decryption should match the raw data rates of the device. Further, any solution whose cost is high is unlikely to be adopted. All of these lead to the following question.

*Is it possible to design a scheme for securing stored data under the following constraints?*

1. *The area and power requirement must be low.*
2. *The actual cost of implementation must be low.*
3. *The speed should match the data rates of the target device.*

## 1.2 Contributions of this work

In this paper, we provide an affirmative answer to the above question through a new construction of a TES (which we call STES) using hardware oriented stream ciphers. Here we provide an overview of the several aspects of the solution. Details are worked out over the rest of the paper.

The application that we have in mind is to encrypt flash memories which have a block-wise organization. Our target applications include USB memories and memory cards like those specified in the SD standard [5]. The SD standard classifies memory cards into four categories based on their speeds. These categories are named as normal speed, high speed, ultra high speed-I (UHS I) and ultra high speed II and the required bus speeds are 12.5 MB/sec; 25 MB/sec; 50-104 MB/sec; and 156-312 MB/sec respectively. The UHS-II category of devices are only recommended for special applications like storing high quality streaming video etc.

These speed requirements are to be contrasted with the speeds of modern hard disks using technologies like serial ATA and native command queuing which can achieve data rates of more than 3 Gigabits/sec (the SATA revision 2 specifies a speed of 6 Gigabits/sec). So, for encryption of SD cards, the speed of encryption is not much demanding, and is much less than the speeds achieved by the TES implementations reported in [15, 31]. As discussed above, speed is only one of the issues. The designs in [15, 31] are neither low area nor low power. Being targeted towards Virtex 4 and 5 FPGAs, these are not also low cost. Hence, in terms of only speed of constrained devices, the designs in [15, 31] are really an over-engineering.

**Low cost designs:** Our target platform is low cost yet performant FPGAs. Examples are the Xilinx Spartan 3 and Lattice ICE40 FPGAs [30]. It may be possible to put such an FPGA in a personal device like a smart phone. TES designs targeted towards such platforms can be directly deployed to small devices.

**Low area/power:** As mentioned earlier, most TES designs are based on block ciphers. Only one work [39] outlines a TES which uses a stream cipher supporting an initialization vector (IV). The stream cipher based construction in [39] has a bug and the corrected construction appears in [38]. This is the starting point of our work. The description in [38, 39] is at a high level using a stream cipher and a hash function. We mention below our choice of the stream cipher, the design of the hash function and the consequent development of a new TES scheme that we call STES. The details of our construction are sufficiently different from that in [38, 39] to necessitate a separate complete security analysis.

The eStream [3] Profile-2 portfolio provides three stream ciphers which have very small hardware footprint, namely Grain128, Mickey 2.0 and Trivium. We consider all these three candidates and later describe implementation results using varying opportunities for parallelism.

The TES construction requires a hash function with provably low collision and differential probabilities. Usual polynomial hash is one way to design such a hash function. But, this requires a finite field multiplier over $GF(2^{\ell})$, where $\ell$ is the IV length of the underlying stream cipher. This may not be a good choice for low area/power designs. Instead, we chose the multi-linear hash function [14,20] for implementation. When used directly, this also requires a $GF(2^{\ell})$ multiplier. We, however, use the so-called Toeplitz version of this hash function, where it is possible to use a $GF(2^{d})$ multiplier where $d$ is a divisor of $\ell$. We call $d$ to be the data path of the hash function. By varying $d$, we can achieve a nice trade-off between the size of the area and the throughput of the hash function. The theoretical possibility of obtaining a hardware efficient hash function using a Toeplitz version of the multi-linear hash was indicated in [35].

Another interesting hash function is based on Winograd's pseudo-dot product [42] and has been mentioned in [9]. Again a direct implementation of this hash function requires a $GF(2^{\ell})$ multiplier, whereas using a Toeplitz version (suggested in [35]) one can use a $GF(2^{d})$ multiplier for some $d$ dividing $\ell$. For a fixed message of a particular length, the total number of multiplications required by the pseudo-dot product based hash function is about half that required by the multi-linear hash function. This seems to suggest that the pseudo-dot product hash function should be the one of choice. Somewhat surprisingly, we show that from the point of view of parallel hardware implementation, there is no significant difference in speed of the two functions. But the pseudo dot product hash functions gives some advantage over the multilinear ones in terms of area when implemented with larger data paths.

Later, we present different design decisions for the hash function and the implementation results. We believe that our implementation of the hash functions is of independent interest and will be useful for other applications. There is no work in the literature which provides such a careful hardware implementation of the multi-linear and the pseudo-dot product hash function as we do.

As mentioned earlier, the stream cipher based TES construction in [39] is at high level. The actual issue of choosing the hash function is not adequately addressed. The multi-linear and the pseudo-dot product hash function that we use requires a long hash key. Due to its length, the key cannot be stored and has to be generated using the stream cipher itself. As a result, the security proof in [39] does not apply any more. We carefully work out the complete proof and obtain a security bound which improves upon the ones given in [38, 39].

The result of all this is that STES is a low power/area design and can be implemented in low cost FPGA. Further, this is achieved while retaining the usual guarantee of rigorous security analysis. To a designer, STES holds out the dual attractiveness of formal security analysis combined with low cost and low area/power implementations.

## 2 Preliminaries

In this section we give an overview of the basic primitives used for the new construction. For a binary string $X$, $\mathsf{bits}(X, i, j)$ denotes the binary string formed by the substring of $X$ extending from position $i$ to position $j$. For binary strings $X$ and $Y$, $X||Y$ denotes the concatenation of $X$ and $Y$. We shall often treat $n$ bit binary strings as elements in $GF(2^n)$, and for $X, Y \in \{0,1\}^n$, $X \oplus Y$ and $X \cdot Y$ (or sometimes $XY$) would mean the addition and multiplication in $GF(2^n)$.

## 2.1 Stream Ciphers with IV

Modern stream ciphers, such as those in the eStream [3] portfolio, take as input a short secret key $K$ and a short initialization vector (IV) and produce a "long" and random looking string of bits. Let $\mathsf{SC}_K : \{0,1\}^\ell \rightarrow \{0,1\}^L$ be a stream cipher with IV, i.e., for every choice of $K$ from a certain pre-defined key space $\mathcal{K}$, $\mathsf{SC}_K$ maps an $\ell$-bit $IV$ to an output string of length $L$ bits. The length $L$ is assumed to be long enough for practical sized messages to be encrypted. For our application, the length of $L$ is determined by the length of the sector. By $\mathsf{SC}^i_K(IV)$ we shall denote the first $i$ bits of the output of $\mathsf{SC}_K(IV)$.

## 2.2 Multilinear Universal Hash

A keyed hash function is chosen from an indexed family $\{H_\tau\}_\tau$ of hash functions. The key $\tau$ is chosen uniformly at random from the index set. Suppose the range consists of $\ell$-bit strings. The hash function is said to be universal if for distinct $X$ and $X'$, $\Pr[H_\tau(X) = H_\tau(X')] = 1/2^\ell$; further it is said to be XOR universal (XU), if for distinct $X$ and $X'$ and any $\ell$-bit string $Y$, $\Pr[H_\tau(X) \oplus H_\tau(X') = Y] = 1/2^\ell$.

We will be interested in a particular type of hash function called the multi-linear function [14, 20]. The following definition is a variant based on the so-called Toeplitz construction. An MLUH (Multilinear Universal Hash) with data path $d$ uses an $(m+b-1)d$-bit key $K$ to map a $dm$-bit message $M$ to a $db$-bit digest. The message $M$ is written as $M = M_1||M_2||\cdots||M_m$ and the key $K$ is written as $K = K_1||K_2||\ldots||K_{m+b-1}$, where each $M_i$ and $K_j$ are $d$ bits long. We define

$$\mathsf{MLUH}^{d,b}_K(M) = h_1||h_2||\cdots||h_b,$$

where

$$\left.\begin{array}{l} h_1 = M_1 \cdot K_1 \oplus M_2 \cdot K_2 \oplus ... \oplus M_m \cdot K_m \\ h_2 = M_1 \cdot K_2 \oplus M_2 \cdot K_3 \oplus ... \oplus M_m \cdot K_{m+1} \\ \;\cdot\;\cdot\;\cdots \\ h_b = M_1 \cdot K_b \oplus M_2 \cdot K_{b+1} \oplus .... \oplus M_m \cdot K_{b+m-1}. \end{array}\right\} \qquad (1)$$

The additions and multiplications are in the field GF$(2^d)$. Note that the message and key lengths are multiples of $d$. This restriction can be lifted by appropriate padding. We do not perform this, since for our application it is easy to ensure that the condition holds.

It is not difficult to show that an $\mathsf{MLUH}$ is an XU function. More specifically for a uniform random key $K$, any pair of distinct messages $M_1, M_2$ and any $\alpha$

$$\Pr_K[\mathsf{MLUH}^{d,b}_K(M_1) \oplus \mathsf{MLUH}^{d,b}_K(M_2) = \alpha] \leq \frac{1}{2^{db}}. \qquad (2)$$

The XOR universal property for a more general version of the multi-linear hash function has been proved in [35].

### 2.3 Pseudo-Dot Product Based Universal Hash

Another construction of hash function can be based on Winograd's pseudo-dot product [42]. This has been pointed out in [9]. We describe the Toeplitz variant of the pseudo-dot product construction as mentioned in [35]. This will be denoted by $\mathsf{PD}$. The PD construction uses an $(m+2b-2)d$-bit key $K$ to map a $dm$-bit message $M$ to a $db$-bit digest. The message $M$ is written as $M = M_1||M_2||\cdots||M_m$ and the key $K$ is written as $K = K_1||K_2||\ldots||K_{m+2b-2}$, where each $M_i$ and $K_j$ are $d$-bit strings. We define

$$\mathsf{PD}_K^{d,b}(M) = h_1||h_2||...||h_b$$

where

$$\left.\begin{aligned}
h_1 &= (M_1 \oplus K_1)(M_2 \oplus K_2) \oplus (M_3 \oplus K_3)(M_4 \oplus K_4) \oplus ... \oplus (M_{m-1} \oplus K_{m-1})(M_m \oplus K_m)\\
h_2 &= (M_1 \oplus K_3)(M_2 \oplus K_4) \oplus (M_3 \oplus K_5)(M_4 \oplus K_6) \oplus ... \oplus (M_{m-1} \oplus K_{m+1})(M_m \oplus K_{m+2})\\
&\cdot\ \cdot\ \cdot\cdot\cdot\\
h_b &= (M_1 \oplus K_{2b-1})(M_2 \oplus K_{2bb}) \oplus ... \oplus (M_{m-1} \oplus K_{m+2b-3})(M_m \oplus K_{m+2b-2})
\end{aligned}\right\} \quad (3)$$

The PD function is also an XU hash function and can be proved by a simple probability calculation.

### 2.4 Tweakable Enciphering Scheme

A tweakable enciphering scheme is a pair of indexed family of functions $(\mathbf{E}_K, \mathbf{D}_K)_{K\in\mathcal{K}}$. For each $K$ in $\mathcal{K}$,

$$\mathbf{E}_K, \mathbf{D}_K : \mathsf{Tweak} \times \mathsf{Msg} \to \mathsf{Cpr}$$

where $\mathsf{Tweak}$, $\mathsf{Msg}$ and $\mathsf{Cpr}$ are non-empty finite sets of binary strings. The set $\mathcal{K}$ is called the key space, $\mathsf{Tweak}$ is the tweak space, $\mathsf{Msg}$ is the message space and $\mathsf{Cpr}$ is the ciphertext space. The functions have to satisfy the following condition. For fixed $K \in \mathcal{K}$ and $T \in \mathsf{Tweak}$,

1. $\mathbf{E}_K(T, \cdot)$ and $\mathbf{D}_K(T, \cdot)$ are length preserving permutations.
2. $\mathbf{D}_K(T, \mathbf{E}_K(T, X)) = X$.

Usually $\mathbf{E}_K(T, X)$ and $\mathbf{D}_K(T, Y)$ are written as $\mathbf{E}_K^T(X)$ and $\mathbf{D}_K^T(Y)$ respectively. $\mathbf{E}$ is called the encryption function and $\mathbf{D}$ is called the decryption function.

The applicability to disk encryption comes via the following relation. For encryption, the tweak $T$ is taken to be the sector address and the message $X$ is taken to be the content of the sector. Let $Y$ be the output of the encryption of $X$ under the tweak $T$ and the secret key $K$. By the length preserving constraint, this $Y$ is of the same length as that of $X$. The value of $X$ is overwritten using $Y$ in the sector pointed to by $T$. Thus, after the whole disk is encrypted, it consists only of the encrypted values. Note that the encryption is done sector-wise and so decryption can also be done sector-wise. This means that the value $Y$ in a particular sector with address $T$ can be decrypted without disturbing the values of the other sectors.

In a typical in-place disk encryption application, the sectors are individually encrypted and stored in the encrypted form. The encryption/decryption module resides just above the disk controller. An encrypted sector read by the disk controller is decrypted by the module and returned to the calling routine. Similarly, the writing of any sector to the disk is first encrypted by the module and then the disk controller writes it to the appropriate sector. This mode of operation ensures that if the disk is accessed without the secret key, then it appears to be random.

In the definition of TES, $\mathsf{Tweak}$, $\mathsf{Msg}$ and $\mathsf{Cpr}$ are mentioned to be non-empty finite sets of binary strings. For disk encryption applications, $\mathsf{Msg} = \mathsf{Cpr}$ and is the set of all binary strings whose lengths are equal to the length of a sector. Similarly, $\mathsf{Tweak}$ is taken to be the set of all binary strings of some fixed length $\ell$ where $2^\ell$ is greater than the number of sectors in a disk.

## 3 Construction of STES

The description of the encryption algorithm using STES is given in Figure 1 and a schematic diagram is shown in Figure 3. The construction is parameterised by a stream cipher $\mathsf{SC}$ supporting $\ell$-bit IVs, a hash function $\mathsf{MLUH}$ with data path $d$ and a fixed $\ell$-bit string $\mathsf{fStr}$. This is emphasized by writing $\mathsf{STES}[\mathsf{SC}, \mathsf{MLUH}, \mathsf{fStr}]$. When one or more of the parameters are clear from the context, we drop these for simplicity of notation; if all three parameters are clear, then we simply write $\mathsf{STES}$. We assume that $d \mid \ell$. Plaintexts and tweaks are fixed length messages. If $P$ is any plaintext and $T$ is any tweak, then we also assume that $d \mid (|P| + |T| - 2\ell)$. For practical implementations, the restrictions on $d$ are easy to ensure as we discuss later.

The secret key for $\mathsf{STES}$ is the secret key $K$ of the underlying stream cipher. In this context, we would like to mention the role that the parameter $\mathsf{fStr}$ can play. From the point of view of the formal security analysis, there is no restriction on $\mathsf{fStr}$. Thus, this can be used as a secret customisation option. In other words, for actual deployment, one may choose a uniform random value for $\mathsf{fStr}$ and keep it secret. This provides an additional layer of obscurity over and above the provable security analysis that we perform. There is another advantage to using $\mathsf{fStr}$ as part of the secret key. The security bound that is obtained is in terms of the IV length $\ell$ and the number of queries for which security holds can be obtained as a function of $2^\ell$. The key length $|K|$ should be at least $\ell$ for the analysis to be meaningful. If the key length is equal to $\ell$, then certain "out of model" attacks may apply as has been pointed out in [18]. Increasing the key length by keeping $\mathsf{fStr}$ as part of the secret key will help in preventing such attacks.

Apart from the secret key $K$, the input to the encryption algorithm of $\mathsf{STES}$ is the tweak $T$ and a plaintext $P$. Similarly, the input to the decryption algorithm of $\mathsf{STES}$ consists of $T$ and the ciphertext $C$.

The encryption algorithm begins with some length calculations, and fixes values for the variables $\ell_1$, $\ell_2$ and $\ell_3$ which determine the key lengths necessary for the different calls to $\mathsf{MLUH}$ made later in the algorithm. Next, the input plaintext is parsed into three parts $P_1$, $P_2$ and $P_3$ where $P_1$ and $P_2$ are both $\ell$ bits long and $P_3$ is $|P| - 2\ell$ bits long. In Line 9, $(\ell_1 + \ell_2)$ bits are generated from the stream cipher $\mathsf{SC}_K$ using the $\mathsf{fStr}$ as input. These bits are parsed into two strings $\tau'$ and $\tau''$ which are later used as keys for $\mathsf{MLUH}$.

The part $P_3$ of the message and the tweak $T$ is hashed using the $\mathsf{MLUH}$ and mixed with the message parts $P_1$ and $P_2$ to generate two strings $A_1$ and $A_2$. These strings are used as an input to the function $\mathsf{Feistel}$ which is described in Figure 2. The function $\mathsf{Feistel}$ receives two keys $K$ and $\tau''$ and it mixes the input strings $A_1$ and $A_2$ by appropriate use of the hash $\mathsf{MLUH}$ and the stream cipher $\mathsf{SC}$. The inverse function for $\mathsf{Feistel}$ is also shown in Figure 2.

**Fig. 1.** STES: A TES using SC and MLUH. The $\ell$-bit string fStr is a parameter to the whole construction. The length of the IV of SC is $\ell$ and the data path of MLUH is $d$.

```
$\textbf{\textit{STES.Encrypt}}_K^T(P)$
1. $b \leftarrow \frac{\ell}{d}$;
2. $b_1 \leftarrow \frac{|P|+|T|-2\ell}{d}$;
3. $\ell_1 \leftarrow (b_1 + b - 1)d$;
4. $\ell_2 \leftarrow (2b - 1)d$;
5. $\ell_3 \leftarrow |P| - 2\ell$;

6. $P_1 \leftarrow \mathsf{bits}(P, 1, \ell)$; /* $|P_1| = \ell$) */
7. $P_2 \leftarrow \mathsf{bits}(P, \ell + 1, 2\ell)$; /* $|P_2| = \ell$ */
8. $P_3 \leftarrow \mathsf{bits}(P, 2\ell + 1, |P|)$; /* $|P_3| = \ell_3$ */

9. $\tau \leftarrow \mathsf{SC}_K^{\ell_1+\ell_2+\ell}(\mathsf{fStr})$;
10. $\tau' \leftarrow \mathsf{bits}(\tau, 1, \ell_1)$;
11. $\beta \leftarrow \mathsf{bits}(\tau, \ell_1 + 1, \ell_1 + \ell)$;
12. $\tau'' \leftarrow \mathsf{bits}(\tau, \ell_1 + \ell + 1, \ell_1 + \ell + \ell_2)$;

13. $Z_1 \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(P_3||T) \oplus \beta$;
14. $A_1 \leftarrow P_1$;
15. $A_2 \leftarrow P_2 \oplus Z_1$;
16. $(B_1, B_2, W) \leftarrow \mathsf{Feistel}_{K,\tau''}^{\ell,d}(A_1, A_2, \ell_3)$;
17. $C_3 \leftarrow P_3 \oplus W$;
18. $Z_2 \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(C_3||T) \oplus (\beta \lll 1)$;
19. $C_1 \leftarrow B_1 \oplus Z_2$;
20. $C_2 \leftarrow B_2$;
$\textbf{\textit{return}}(C_1||C_2||C_3)$;
```

```
$\textbf{\textit{STES.Decrypt}}_K^T(C)$
1. $b \leftarrow \frac{\ell}{d}$;
2. $b_1 \leftarrow \frac{|C|+|T|-2\ell}{d}$;
3. $\ell_1 \leftarrow (b_1 + b - 1)d$;
4. $\ell_2 \leftarrow (2b - 1)d$;
5. $\ell_3 \leftarrow |C| - 2\ell$;

6. $C_1 \leftarrow \mathsf{bits}(C, 1, \ell)$; /* $|C_1| = \ell$) */
7. $C_2 \leftarrow \mathsf{bits}(C, \ell + 1, 2\ell)$; /* $|C_2| = \ell$ */
8. $C_3 \leftarrow \mathsf{bits}(C, 2\ell + 1, |C|)$; /* $|C_3| = \ell_3$ */

9. $\tau \leftarrow \mathsf{SC}_K^{\ell_1+\ell_2+\ell}(\mathsf{fStr})$;
10. $\tau' \leftarrow \mathsf{bits}(\tau, 1, \ell_1)$;
11. $\beta \leftarrow \mathsf{bits}(\tau, \ell_1 + 1, \ell_1 + \ell)$;
12. $\tau'' \leftarrow \mathsf{bits}(\tau, \ell_1 + \ell + 1, \ell_1 + \ell + \ell_2)$;

13. $Z_2 \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(C_3||T) \oplus (\beta \lll 1)$;
14. $B_1 \leftarrow C_1 \oplus Z_2$;
15. $B_2 \leftarrow C_2$;
16. $(A_1, A_2, W) \leftarrow \mathsf{InvFeistel}_{K,\tau''}^{\ell,d}(B_1, B_2, \ell_3)$;
17. $P_3 \leftarrow C_3 \oplus W$;
18. $Z_1 \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(P_3||T) \oplus \beta$;
19. $P_1 \leftarrow A_1$;
20. $P_2 \leftarrow A_2 \oplus Z_1$;
$\textbf{\textit{return}}(P_1||P_2||P_3)$;
```

**Fig. 2.** The Feistel network (and its inverse) constructed using a stream cipher and a MLUH. The variable $\ell$ is the length of an IV for SC and $d$ is the data path of MLUH. This definition is different from the usual Feistel construction: a positive integer $i$ is provided as an additional input and a binary string $W$ of length $i$ is returned as an additional output.

```
$\mathsf{Feistel}_{K,\tau''}(A_1, A_2, i)$
1. $H_1 \leftarrow \mathsf{MLUH}_{\tau''}^{d,b}(A_1)$;
2. $F_1 \leftarrow H_1 \oplus A_2$;
3. $(G_1, W) \leftarrow \mathsf{SC}_K^{\ell+i}(F_1)$;
4. $F_2 \leftarrow A_1 \oplus G_1$;
5. $G_2 \leftarrow \mathsf{SC}_K^{\ell}(F_2)$;
6. $B_2 \leftarrow F_1 \oplus G_2$;
7. $H_2 \leftarrow \mathsf{MLUH}_{\tau''}^{d,b}(B_2)$;
8. $B_1 \leftarrow H_2 \oplus F_2$;
$\textbf{\textit{return}}(B_1, B_2, W)$;
```

```
$\mathsf{InvFeistel}_{K,\tau''}(B_1, B_2, i)$
1. $H_2 = \mathsf{MLUH}_{\tau''}^{d,b}(B_2)$;
2. $F_2 = H_2 \oplus B_1$;
3. $G_2 = \mathsf{SC}_K^{\ell}(F_2)$;
4. $F_1 = B_1 \oplus G_2$;
5. $(G_1, W) = \mathsf{SC}_K^{\ell+i}(F_1)$;
6. $A_1 = F_2 \oplus G_1$;
7. $H_1 = \mathsf{MLUH}_{\tau''}^{d,b}(A_1)$;
8. $A_2 \leftarrow H_1 \oplus F_1$;
$\textbf{\textit{return}}(A_1, A_2, W)$;
```

The call to Feistel also returns the string $W$ of length equal to $\ell_3$ which is the length of $P_3$. This string $W$ is XORed with $P_3$ to obtain $C_3$. The Feistel network produces two more $\ell$ bit outputs $B_1$ and $B_2$. $B_1$ and $B_2$ are used to produce the ciphertexts $C_1$ and $C_2$ respectively.

From the description given in Figures 1 and 2, it may appear that the string $W$ of length $\ell_3$ is required to be actually returned to the main body of the algorithm. This, however, is not the case. For example, depending on the specific design choices it may be possible to compute $W$ (in line 3, Fig. 2) and $Z_2$ (in line 19, Fig. 1) in parallel.

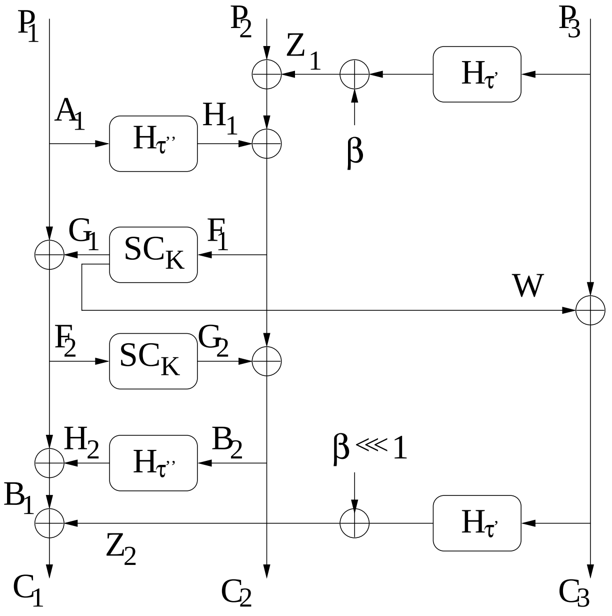

**Fig. 3.** Schematic diagram of the encryption algorithm of STES.

## 3.1 Variant of STES Using PD

The description of the algorithm STES can be modified by using the pseudo dot-product hash PD which is described in Section 2.2. If PD is used instead of MLUH the key lengths required are to be suitably changed. For hashing $m$ blocks (where each block is $d$ bits long) of message the PD construction requires $m + 2b - 2$ blocks of keys, where $bd$ is the length of the output. Hence the parameter $\ell_1$ in Line 4 must be fixed to $b_1 + 2b - 2$ and $\ell_2$ to $(4b - 2)d$. All the calls to MLUH should be replaced by PD in the algorithm STES and in the function Feistel with the same parameters as it appears in the descriptions. This variant is formally denoted as $\mathsf{STES}[\mathsf{SC}, \mathsf{PD}, \mathsf{fStr}]$.

## 3.2 Some Characteristics of the Construction

**Hash keys:** The hash keys $\tau'$ and $\tau''$ are generated using call $\mathsf{SC}_K(\mathsf{fStr})$. If it is possible to store $\tau'$ and $\tau''$, then the call to SC in Line 9 is not required and this will gain some efficiency. But, as the key $\tau'$ is much larger in size compared to $K$, for most applications this will not be practical. Using a register to store this key within the FPGA will greatly push up the area requirement. For designs targeted at small area implementations, it is necessary to generate the hash key on the fly.
**Message length:** STES is defined only for fixed length messages. This is because our intended application is encryption of disks and flash memories where the message is a sector and has a fixed

length. It is possible to extend the construction to accommodate variable length messages as has been done in [39]. This, however, results in slightly more computation than the fixed length case. Since, variable length messages is not required in our context, we chose to cut out the additional complexity of padding and formatting from the algorithm.

**Efficiency:** The computationally costly operations that take place in the algorithm are the calls to the stream cipher and the hash functions. There is one call to the stream cipher from the main body of the algorithm to generate the hash keys and the other two calls are part of the function $\mathsf{Feistel}$. It is to be noted that in real life stream ciphers are quite fast in generation of the outputs, but when a stream cipher is called on different initialization vectors then there is a significant time required for initialization. The three calls to the stream cipher required in STES are all on different initialization vectors. Hence, stream cipher initializations occupy a significant amount of the time required for STES.

The hash functions $\mathsf{MLUH}$ and $\mathsf{PD}$ can be implemented very efficiently in hardware with a proper choice of the data path $d$. The choice of $d$ dictates the amount of parallelism possible. Recall that the main goal of the construction is to enable a hardware realization which uses small amount of hardware resources. A proper choice of the stream cipher and the data path can help in realizing a circuit with adequate throughput but with a small hardware footprint. These issues are discussed in details in Section 6 where we demonstrate that STES meets the expected efficiency requirements both in terms of time and circuit size.

**Parallelism:** There exist ample scope to exploit parallelism in the construction of STES. In the hardware implementation that we present later, we decided to use two stream cipher cores, our specific design choices give rise to an architecture where decryption is a bit faster than encryption. This characteristic of the design has some good practical implications, as read speeds in memory are faster than the write speeds and it is expected that a typical block would be read many more times than it would be written.

## 4 Security of STES

In this section, we state the usual security theorem for STES. To do this, we need to introduce the appropriate notions of security of a stream cipher with IV and that of a tweakable enciphering scheme.

### 4.1 Pseudo-Random Function

Let $\mathsf{Dom}$ and $\mathsf{Ran}$ be two non-empty finite sets of binary strings. Let $f_K$ be an indexed set of functions where $f_K : \mathsf{Dom} \rightarrow \mathsf{Ran}$ and $K$ is chosen from some index set $\mathcal{K}$. The notion of pseudo-random function (PRF) is formalized in the following manner.

Let $K$ be chosen uniformly at random from $\mathcal{K}$. An adversary $\mathcal{A}$ has to distinguish $f_K$ from a uniform random function $f^*$ where $f^*$ is chosen uniformly at random from the set of all functions from $\mathsf{Dom}$ to $\mathsf{Ran}$. $\mathcal{A}$ is a probabilistic algorithm which has oracle access to either $f_K$ or $f^*$. Suppose the oracle is $f_K$. The adversary $\mathcal{A}$ submits queries $X_1, \ldots, X_q$ to the oracle and gets back the responses $f_K(X_1), \ldots, f_K(X_q)$. $\mathcal{A}$ can make the queries in an adaptive manner, i.e., it can decide on the $i$-th query after receiving the responses to the first $(i-1)$ queries. Without loss of generality, we will assume that the queries are distinct. At the end of the interaction, $\mathcal{A}$ outputs a bit.

Let $\Pr[\mathcal{A}^{f_K} \Rightarrow 1]$ denote the probability that $\mathcal{A}$ outputs 1 after interacting with $f_K$. This probability is over the randomness of $f_K$ (arising from the random choice of $K$) as well as the randomness of $\mathcal{A}$.

Similarly, let $\mathcal{A}$ interact with $f^*$ and let $\Pr[\mathcal{A}^{f^*} \Rightarrow 1]$ denote the probability that $\mathcal{A}$ outputs 1 after interacting with $f^*$. The advantage of $\mathcal{A}$ in distinguishing $f_K$ from the uniform random $f^*$ is defined as follows.

$$\mathsf{Adv}_f^{\mathrm{prf}}(\mathcal{A}) = \Pr[\mathcal{A}^{f_K} \Rightarrow 1] - \Pr[\mathcal{A}^{f^*} \Rightarrow 1]. \tag{4}$$

Let $\mathsf{Adv}_f^{\mathrm{prf}}(t, q, \sigma)$ be the supermum of the advantages of all adversaries running in time $t$, making $q$ queries and providing a total of $\sigma$ bits in all its queries. The quantity $\sigma$ is called the query complexity. Note that $\mathsf{Adv}_f^{\mathrm{prf}}(t, q, \sigma)$ is always positive even though the quantity defined in (4) can sometimes be negative. The value of $\mathsf{Adv}_f^{\mathrm{prf}}(t, q, \sigma)$ is called the PRF-bound for $f$.

### 4.2 Stream Cipher With IV

Recall that for a key $K$ from the key space $\mathcal{K}$, a stream cipher with IV is a function $\mathsf{SC}_K : \{0,1\}^\ell \to \{0,1\}^L$. The basic idea of security is that for a uniform random $K$ and for distinct inputs $IV_1, \ldots, IV_q$, the strings $\mathsf{SC}_K(IV_1), \ldots, \mathsf{SC}_K(IV_q)$ should appear to be independent and uniform random to an adversary. This is formalised by requiring a stream cipher to be a PRF. See [8] for further discussion on this issue. $\mathsf{Adv}_{\mathsf{SC}}^{\mathrm{prf}}(t, q, \sigma)$ denotes the PRF-advantage of $\mathsf{SC}$ against any adversary that runs in time $t$, makes $q$ queries and has query complexity $\sigma$.

### 4.3 Tweakable Enciphering Scheme

Consider a $\mathsf{TES} = (\mathbf{E}_K, \mathbf{D}_K)_{K \in \mathcal{K}}$. Let $K$ be chosen uniformly at random and an adversary $\mathcal{A}$ is given access to the oracles $(\mathbf{E}_K, \mathbf{D}_K)$. A query to $\mathbf{E}_K$ is a pair $(T, X)$ and a query to $\mathbf{D}_K$ is a pair $(T, Y)$, where $T$ is a tweak, $X$ is a message and $Y$ is a ciphertext. Appropriate responses to the queries are provided to the adversary $\mathcal{A}$.

Note that $\mathcal{A}$ is allowed to make the queries in an adaptive manner, i.e., for the $i$-th query it can decide on whether to send it to $\mathbf{E}_K$ or $\mathbf{D}_K$ and the content of the query based on the responses it has received to the previous $(i-1)$ queries. The restrictions are that no two queries to $\mathbf{E}_K$ should be equal; no two queries to $\mathbf{D}_K$ should be equal; if $Y$ has been obtained as a response to a query $(T, X)$ to $\mathbf{E}_K$, then the query $(T, Y)$ to $\mathbf{D}_K$ is not allowed; and similarly, if $X$ has been obtained as a response to a query $(T, Y)$ to $\mathbf{D}_K$, then the query $(T, X)$ to $\mathbf{E}_K$ is not allowed. These queries are pointless as the adversary already knows the answer to these queries. Let $\Pr[\mathcal{A}^{\mathbf{E}_K, \mathbf{D}_K} \Rightarrow 1]$ be the probability that $\mathcal{A}$ outputs 1 after interacting with the oracles $\mathbf{E}_K$ and $\mathbf{D}_K$.

For each tweak $T$, let $\Pi(T, \cdot)$ be a length preserving permutation chosen uniformly at random from the set of all length preserving permutations from $\mathsf{Dom}$ to $\mathsf{Ran}$. By $\Pi$ we denote the collection $\{\Pi(T, \cdot)\}_{T \in \mathsf{Tweak}}$ of all these tweak-indexed uniform random length preserving permutations. For each $T$, let $\Pi^{-1}(T, \cdot)$ be the inverse of $\Pi(T, \cdot)$ and let $\Pi^{-1}$ be the collection $\{\Pi^{-1}(T, \cdot)\}_{T \in \mathsf{Tweak}}$.

Suppose $\mathbf{E}_K$ and $\mathbf{D}_K$ are replaced by $\Pi$ and $\Pi^{-1}$ respectively and consider the interaction of $\mathcal{A}$ with $\Pi$ and $\Pi^{-1}$. Let $\Pr[\mathcal{A}^{\Pi,\Pi^{-1}} \Rightarrow 1]$ be the probability that $\mathcal{A}$ outputs 1 after such interaction. The advantage of $\mathcal{A}$ is defined as follows.

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{TES}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathbf{E}_K,\mathbf{D}_K} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(\cdot,\cdot),\$(\cdot,\cdot)} \Rightarrow 1]. \tag{5}$$

Define $\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{TES}}(t,q,\sigma)$ to be the supremum of the advantages of all adversaries which run in time $t$, make a total of $q$ queries and send a total of $\sigma$ bits in all the queries. Security of a scheme against an adversary which has access to both the encryption and the decryption oracles is called security as a strong pseudo-random permutation.

Suppose now that the oracles $\mathbf{E}_K$ and $\mathbf{D}_K$ are replaced by two oracles which return independent and uniform random strings on any input. More precisely, if $(T, X)$ is a query to $\mathbf{E}_K$, then an independent and uniform random string of length equal to the length of $X$ is returned; if $(T, Y)$ is a query to $\mathbf{D}_K$, then similarly, an independent and uniform random string of length equal to the length of $Y$ is returned. Let $\Pr[\mathcal{A}^{\$(\cdot,\cdot),\$(\cdot,\cdot)} \Rightarrow 1]$ be the probability that $\mathcal{A}$ outputs 1 after such interaction. The advantage of $\mathcal{A}$ is defined as follows.

$$\mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathbf{E}_K,\mathbf{D}_K} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(\cdot,\cdot),\$(\cdot,\cdot)} \Rightarrow 1]. \tag{6}$$

Define $\mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(t,q,\sigma)$ to be the supremum of all advantages of adversaries which run in time $t$, make a total of $q$ queries and send a total of $\sigma$ bits in all the queries.

The $\pm\mathrm{rnd}$ and $\pm\widetilde{\mathrm{prp}}$ advantages are related as follows.

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{TES}}(\mathcal{A}) \leq \mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{TES}}(\mathcal{A}) + \binom{q}{2}\frac{1}{2^{\ell}}. \tag{7}$$

For a proof of (7) see [16,27].

## 4.4 Security Statement for STES

The following theorem specifies the security of STES.

**Theorem 1.** *Let* $\mathsf{SC}_K : \{0,1\}^{\ell} \to \{0,1\}^{L}$ *be a stream cipher with IV and* $\mathsf{H}$ *be one of the hash functions* $\mathsf{MLUH}$ *or* $\mathsf{PD}$*. For* $\sigma \geq q \geq 1$ *and* $t \geq 1$,

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES[SC,H,fStr]}}(t,q,\sigma) \leq \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{SC}}(t',q,\sigma) + \frac{9q^2}{2^{\ell+1}}. \tag{8}$$

*Here* $t'$ *is* $t + t''$*, where* $t''$ *is the time to process* $q$ *queries using* $\mathsf{STES[SC,H,fStr]}$.

The theorem guarantees that if the stream cipher acts like a random function then, for any arbitrary adversary which makes a reasonable number of queries, the advantage of distinguishing $\mathsf{STES[SC]}$ from a tweak-indexed family of length preserving permutations is small. The proof of the theorem consists of a standard game transition argument and a combinatorial analysis of some collision probabilities. The proof is given in Section 5.

## 5 Proof of Theorem 1

In the proof, the string $\mathsf{fStr}$ will be fixed and so we will not explicitly mention this as a parameter to $\mathsf{STES}$. The analysis of the proof will be done assuming the hash function $\mathsf{H}$ to be $\mathsf{MLUH}$. Essentially the same analysis also holds when $\mathsf{H}$ is instantiated as $\mathsf{PD}$.

Let $\delta$ be a uniform random function from $\{0,1\}^\ell$ to $\{0,1\}^L$, i.e., $\delta$ is chosen uniformly at random from the set of all functions from $\{0,1\}^\ell$ to $\{0,1\}^L$. This means that for distinct inputs $X_1,\ldots,X_q$, the values $\delta(X_1),\ldots,\delta(X_q)$ are independent and uniform random. This property will be used in the argument below. In the first step of the proof, the stream cipher $\mathsf{SC}$ is replaced by $\delta$. Note that this is only a conceptual step and there is no need to obtain the actual construction. In fact, there is no need to even chose the whole of $\delta$ and its behavior can be simulated in an incremental fashion as follows. Keep a list of hitherto generated pairs of inputs and outputs; for any input, one needs to check whether it has already occurred and if so, return the corresponding output; if not, return an independent and uniform random string.

Denote by $\mathsf{STES}[\delta,\mathsf{H}]$ the corresponding construction. It is quite standard to argue that the following inequality holds.

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\mathsf{SC},\mathsf{H}]}(t,q,\sigma) \leq \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{SC}}(t',q,\sigma) + \mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta,\mathsf{H}]}(t,q,\sigma). \tag{9}$$

The idea is that if there is an adversary which can distinguish between $\mathsf{STES}[\mathsf{SC}]$ and $\mathsf{STES}[\delta]$, then that adversary can be used to distinguish between $\mathsf{SC}$ and a uniform random function and so breaks the PRF-property of $\mathsf{SC}$. The parameters $q$ and $\sigma$ carry over directly whereas the parameter $t$ increases to $t'$ since during the simulation, each query has to be processed using either the encryption or the decryption algorithm of $\mathsf{STES}[\mathsf{SC},\mathsf{H}]$.

The construction $\mathsf{STES}[\delta]$ is parameterized by the uniform random function $\delta$. Unlike $\mathsf{SC}$, there is no computational assumption on $\delta$. Basically, the computational aspect is taken care of by the bound on the PRF-advantage of $\mathsf{SC}$. So, the rest of the proof proceeds in an information theoretic manner. The time parameter in $\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(t,q,\sigma)$ is redundant, since allowing unlimited time does not help the adversary. So, we drop $t$ and use the notation $\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(q,\sigma)$. The main part of the proof is to show the following.

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(t,q,\sigma) \leq \frac{9q^2}{2^{\ell+1}}. \tag{10}$$

From (7), we have

$$\mathsf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathsf{STES}[\delta]}(q,\sigma) \leq \mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{STES}[\delta]}(\sigma) + \binom{q}{2}\frac{1}{2^\ell}. \tag{11}$$

So, the task reduces to upper bounding $\mathsf{Adv}^{\pm\mathrm{rnd}}_{\mathsf{STES}[\delta]}$. In other words, this is to show that the advantage of an adversary in distinguishing $\mathsf{STES}[\delta]$ from oracles which simply return independent and uniform random strings is small. The proof now proceeds via a sequence of games as described below.

In each game, the adversary $\mathcal{A}$ makes a total of $q$ queries. For convenience of description, we introduce a variable $ty^s$ for each $s=1,\ldots,q$. The value of $ty^s = enc$ (resp. $ty^s = dec$) denotes the corresponding query to be an encryption (resp. decryption) query. At the end of each game, the adversary outputs a bit. By $\Pr[\mathcal{A}^{\mathbf{G}} \Rightarrow 1]$ we denote the event that the adversary outputs 1 in Game $\mathbf{G}$ where $\mathbf{G}$ is one of $\mathbf{G0}$, $\mathbf{G1}$ or $\mathbf{G2}$.

The first game $\mathbf{G0}$ is depicted in Figure 4. Game $\mathbf{G0}$ is just the rewrite of the algorithm of $\mathsf{STES}$ in Fig. 1, but we replace the stream cipher $\mathsf{SC}$ by a uniform random function $\delta$. The random function is constructed on the fly using the subroutine $\delta()$, which is also shown in Figure 4. The subroutine $\delta()$ maintains a table $T$, indexed on the strings in $\{0,1\}^{\ell}$ and is initially undefined everywhere. In the table $T[\,]$ the subroutine keeps information regarding the values returned by it corresponding to the inputs. When called on an input $X \in \{0,1\}^{\ell}$, $\delta()$ checks if $T[X]$ is undefined; if so, then it returns a random string in $\{0,1\}^{L}$ and stores the returned value in $T[X]$; otherwise, it returns $T[X]$ and sets a flag labeled $\mathsf{bad}$ to true. Note that game $\mathbf{G0}$ is a perfect simulation of $\mathsf{STES}$ instantiated with the random function $\delta$. Hence, if $\mathcal{A}$ is the adversary interacting with $\mathbf{G0}$, and if we denote the encryption and decryption procedures of $\mathsf{STES}[\delta]$ as $\Pi_\delta$ and $\Pi_\delta^{-1}$ respectively, then we have

$$\Pr[\mathcal{A}^{\mathbf{G0}} \Rightarrow 1] = \Pr[\mathcal{A}^{\Pi_\delta(.,.),\Pi_\delta^{-1}(.,.)} \Rightarrow 1]. \tag{12}$$

We change game $\mathbf{G0}$ to game $\mathbf{G1}$ by eliminating the boxed entry in game $\mathbf{G0}$. $\mathbf{G1}$ is shown in Figure 4. With this change, the games $\mathbf{G0}$ and $\mathbf{G1}$ executes in the same manner unless the $\mathsf{bad}$ flag is set to true. Hence, using the difference Lemma (see [7,40])

$$|\Pr[\mathcal{A}^{\mathbf{G0}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathbf{G1}} \Rightarrow 1]| \le \Pr[\mathcal{A}^{\mathbf{G1}} \text{ sets } \mathsf{bad}]. \tag{13}$$

In Game $\mathbf{G1}$, the responses received by $\mathcal{A}$ are random strings as $C_1$, $C_2$ and $C_3$ are all outputs of $\delta()$ xor-ed with other independent strings. Also, in Game $\mathbf{G1}$, $\delta()$ responds with random strings irrespective of the inputs it receives.

Now, we do a purely syntactic change to $\mathbf{G1}$ to obtain $\mathbf{G2}$ which is shown in Figure 5. In $\mathbf{G2}$, when an encryption or decryption query from $\mathcal{A}$ is received, a random string of the length equal to that of the message/cipher length is returned immediately. After all the $q$ queries of the adversary have been answered, the game enters the finalization phase. The finalization of $\mathbf{G2}$ runs in two phases. In the first phase, based on the query and the response, the internal random variables in the algorithm are adjusted and these values are inserted in the $\mathcal{D}$. In Phase 2, if there is a collision within $\mathcal{D}$, i.e., two random variables in $\mathcal{D}$ take the same value, then the bad flag is set to true.

As $\mathbf{G1}$ and $\mathbf{G2}$ provide the same view to the adversary and differ only in the way they are written, we have

$$\Pr[\mathcal{A}^{\mathbf{G1}} \Rightarrow 1] = \Pr[\mathcal{A}^{\mathbf{G2}} \Rightarrow 1] \text{ and } \Pr[\mathcal{A}^{\mathbf{G1}} \text{ sets } \mathsf{bad}] = \Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}]. \tag{14}$$

Moreover, when $\mathcal{A}$ interacts with $\mathbf{G2}$ it gets random strings as responses to all its queries, and so

$$\Pr[\mathcal{A}^{\mathbf{G2}} \Rightarrow 1] = \Pr[\mathcal{A}^{\$(.,.),\$(.,.)} \Rightarrow 1]. \tag{15}$$

Hence, using (13), (14) and (15), we have

$$\begin{aligned}\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{STES}[\delta]}(\mathcal{A}) &= \Pr[\mathcal{A}^{\Pi_\delta(.,.),\Pi_\delta^{-1}(.,.)} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(.,.),\$(.,.)} \Rightarrow 1] \\ &\le \Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}]. \end{aligned} \tag{16}$$

```
Subroutine $\delta(X)$
01.    $Y \xleftarrow{\$} \{0,1\}^L$;
02.    if $X \in \mathcal{D}$ then $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{Y \leftarrow T[X]}$; end if;
03.    $T[X] \leftarrow Y$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{X\}$;
04.    return $Y$;
Initialization:
    for all $X \in \{0,1\}^\ell$, $T[X] \leftarrow \mathsf{undef}$; endfor
    $\mathsf{bad} \leftarrow \mathsf{false}$; $\mathcal{D} \leftarrow \{\mathsf{fStr}\}$;
        $\tau \leftarrow \delta(\mathsf{fStr})$;
        $\tau' \leftarrow \mathsf{bits}(\tau, 1, \ell_1)$;
        $\tau'' \leftarrow \mathsf{bits}(\tau, \ell_1 + 1, \ell_1 + \ell_2)$;
        $\beta \leftarrow \mathsf{bits}(\tau, \ell_1 + \ell_2 + 1, \ell_1 + \ell_2 + \ell)$;
```

```
$\mathsf{Feistel}_{K,\tau''}(A_1^s, A_2^s, \ell_3)$
201.   $b^s \leftarrow \lceil \frac{\ell}{d} \rceil$
202.   $H_1^s \leftarrow \mathsf{MLUH}_{\tau''}^{d,b}(A_1^s)$;
203.   $F_1^s \leftarrow H_1^s \oplus A_2^s$;
204.   $(G_1^s, W^s) \leftarrow \mathsf{bits}(\delta(F_1^s), 1, \ell + \ell_3)$;
205.   $F_2^s \leftarrow A_1^s \oplus G_1^s$;
206.   $G_2^s \leftarrow \mathsf{bits}(\delta(F_2^s), 1, \ell)$;
207.   $B_2^s \leftarrow F_1^s \oplus G_2^s$;
208.   $H_2^s \leftarrow \mathsf{MLUH}_{\tau''}^{d,b}(B_2^s)$;
209.   $B_1^s \leftarrow H_2^s \oplus F_2^s$;
210.   return$(B_1^s, B_2^s, W^s)$;
```

```
$\mathsf{InvFeistel}_{K,\tau''}(B_1^s, B_2^s, \ell_3)$
201.   $b^s \leftarrow \lceil \frac{\ell}{d} \rceil$;
202.   $H_2^s = \mathsf{MLUH}_{\tau''}^{d,b}(B_2^s)$;
203.   $F_2^s = H_2^s \oplus B_1^s$;
204.   $G_2^s = \mathsf{bits}(\delta(F_2^s), 1, \ell)$;
205.   $F_1^s = B_1^s \oplus G_2^s$;
206.   $(G_1^s, W^s) = \mathsf{bits}(\delta(F_1^s), 1, \ell + \ell_3)$;
207.   $A_1^s = F_2^s \oplus G_1^s$;
208.   $H_1^s = \mathsf{MLUH}_{\tau''}^{d,b}(A_1^s)$;
209.   $A_2^s \leftarrow H_1^s \oplus F_1^s$;
210.   return$(A_1^s, A_2^s, W^s)$;
```

Response to the $s^{th}$ query:

```
Case $ty^s = enc$:
100.   $P_1^s \leftarrow \mathsf{bits}(P^s, 1, \ell)$; /* $|P_1| = \ell)$ */
101.   $P_2^s \leftarrow \mathsf{bits}(P^s, \ell + 1, 2\ell)$; /* $|P_2| = \ell$ */
102.   $P_3^s \leftarrow \mathsf{bits}(P^s, 2\ell + 1, |P|)$; /* $|P_3| = \ell_3$ */

103.   $Z_1^s \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s || T^s) \oplus \beta$;
104.   $A_1^s \leftarrow P_1^s$;
105.   $A_2^s \leftarrow P_2^s \oplus Z_1^s$;
106.   $(B_1^s, B_2^s, W^s) \leftarrow \mathsf{Feistel}_{K,\tau''}^{\ell,d}(A_1^s, A_2^s, \ell_3)$;
107.   $C_3^s \leftarrow P_3^s \oplus W^s$;
108.   $Z_2^s \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s || T^s) \oplus (\beta \lll 1)$;
109.   $C_1^s \leftarrow B_1^s \oplus Z_2^s$;
110.   $C_2^s \leftarrow B_2^s$;
return$(C_1^s || C_2^s || C_3^s)$;
```

```
Case $ty^s = dec$:
100.   $C_1^s \leftarrow \mathsf{bits}(C^s, 1, \ell)$; /* $|P_1| = \ell)$ */
101.   $C_2^s \leftarrow \mathsf{bits}(C^s, \ell + 1, 2\ell)$; /* $|P_2| = \ell$ */
102.   $C_3^s \leftarrow \mathsf{bits}(C^s, 2\ell + 1, |P|)$; /* $|P_2| = \ell_3$ */

103.   $Z_2^s \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s || T^s) \oplus (\beta \lll 1)$;
104.   $B_1^s \leftarrow C_1^s \oplus Z_1^s$;
105.   $B_2^s \leftarrow C_2^s$;
106.   $(A_1^s, A_2^s, W^s) \leftarrow \mathsf{InvFeistel}_{K,\tau''}^{\ell,d}(A_1^s, A_2^s, \ell_3)$;
107.   $P_3^s \leftarrow C_3^s \oplus W^s$;
108.   $Z_1^s \leftarrow \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s || T^s) \oplus \beta$;
109.   $P_1^s \leftarrow A_1^s$;
110.   $P_2^s \leftarrow A_2^s \oplus Z_1^s$;
return$(P_1^s || P_2^s || P_3^s)$;
```

**Fig. 4.** Games **G0** and **G1**. The full description is of Game **G0**; Game **G1** is obtained by removing the boxed entry in Line **02**.

## 5.1 Collision Analysis

The rest of the proof is devoted to computing a bound on $\Pr[\mathcal{A}^{\mathbf{G2}}$ sets $\mathsf{bad}]$. Here $\mathcal{A}$ is an arbitrary adversary which asks $q$ queries each consisting of a message/cipher of length $m\ell$ bits and a tweak of $\ell$ bits. If $\mathsf{COLLD}$ denotes the event that there is a collision in $\mathcal{D}$ as described in the Game **G2**, then

$$\Pr[\mathcal{A}^{\mathbf{G2}} \text{ sets } \mathsf{bad}] = \Pr[\mathsf{COLLD}]. \tag{17}$$

```
Initialization:
$\mathcal{D} \xleftarrow{\$} \{\mathsf{fStr}\}$; $\tau' \xleftarrow{\$} \{0,1\}^{\ell_1}$; $\tau'' \xleftarrow{\$} \{0,1\}^{\ell_2}$; $\beta \xleftarrow{\$} \{0,1\}^{\ell}$;
Response to the $s^{th}$ query:

Case $ty^s = enc$:
101.  $C^s \xleftarrow{\$} \{0,1\}^{|P|}$
102.  $C_1^s \leftarrow \mathsf{bits}(C^s, 1, \ell)$;
103.  $C_2^s \leftarrow \mathsf{bits}(C^s, \ell+1, 2\ell)$;
104.  $C_3^s \leftarrow \mathsf{bits}(C^s, 2\ell+1, |P|)$;
return$(C_1^s||C_2^s||C_3^s)$;

Case $ty^s = dec$:
101.  $P^s \xleftarrow{\$} \{0,1\}^{|C|}$
102.  $P_1^s \leftarrow \mathsf{bits}(P^s, 1, \ell)$;
103.  $P_2^s \leftarrow \mathsf{bits}(P^s, \ell+1, 2\ell)$;
104.  $P_3^s \leftarrow \mathsf{bits}(P^s, 2\ell+1, |C|)$;
return$(P_1^s||P_2^s||P_3^s)$;

Finalization:

FIRST PHASE
for $s \leftarrow 1$ to $q$ do
    Case $ty^s = enc$:
        $F_1^s \leftarrow P_2^s \oplus \beta \oplus \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s||T^s) \oplus \mathsf{MLUH}_{\tau''}^{d,b}(P_1^s)$;
        $F_2^s \leftarrow C_1^s \oplus (\beta \lll 1) \oplus \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s||T^s) \oplus \mathsf{MLUH}_{\tau''}^{d,b}(C_2^s)$;
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{F_1^s, F_2^s\}$;
    Case $ty^s = dec$:
        $F_2^s \leftarrow C_1^s \oplus (\beta \lll 1) \oplus \mathsf{MLUH}_{\tau'}^{d,b}(C_3^s||T^s) \oplus \mathsf{MLUH}_{\tau''}^{d,b}(C_2^s)$;
        $F_1^s \leftarrow P_2^s \oplus \beta \oplus \mathsf{MLUH}_{\tau'}^{d,b}(P_3^s||T^s) \oplus \mathsf{MLUH}_{\tau''}^{d,b}(P_1^s)$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{F_1^s, F_2^s\}$;
endfor;

SECOND PHASE
bad $\leftarrow$ false;
if (two variables in $\mathcal{D}$ are equal) then bad $\leftarrow$ true;
```

**Fig. 5.** Game **G2**.

We shall now concentrate on finding an upper bound for $\Pr[\mathsf{COLLD}]$. The following simple result states a property of rotation that we will require later.

**Lemma 1.** *If $\beta$ is chosen uniformly at random from $\{0,1\}^\ell$ and $X$ is any $\ell$-bit string, then* $\Pr[\beta \oplus (\beta \lll 1) = X] = 1/2^{\ell-1}$.

The set $\mathcal{D}$ is as follows:

$$\mathcal{D} = \{F_1^s, F_2^s : 1 \le s \le q\} \cup \{\mathsf{fStr}\}.$$

Here $\mathsf{fStr}$ is a fixed string while the other elements of $\mathcal{D}$ are random variables. All variables in $\mathcal{D}$ are distributed over $\{0,1\}^\ell$. As mentioned earlier, the proof is using $\mathsf{MLUH}_\tau^{d,b}(X)$ to instantiate $H_\tau(X)$, Essentially the same arguments hold when the $\mathsf{PD}$ construction is used to instantiate $H$. The variables in $\mathcal{D}$ can be expressed in terms of the plaintext and ciphertext blocks as follows.

$$F_1^s = P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s) \oplus H_{\tau''}(P_1^s),$$
$$F_2^s = C_1^s \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^s||T^s) \oplus H_{\tau''}(C_2^s).$$

Noting the following points will help in following the analysis.

1. In Game **G2**, the string $\tau = \tau'||\tau''||\beta$ is selected uniformly at random and is independent of all other variables.

2. If $ty^s = enc$, then $(C_1^s, C_2^s, C_3^s)$ is uniformly distributed and independent of all other variables; if $ty^s = dec$, then $(P_1^s, P_2^s, P_3^s)$ is uniformly distributed and independent of all other variables.
3. In response to each query, $\mathcal{A}$ receives either $(C_1^s, C_2^s, C_3^s)$ or $(P_1^s, P_2^s, P_3^s)$. These variables are independent of $\tau$ and so the queries made by $\mathcal{A}$ are also independent of $\tau$.

Using the randomness of $\beta$, the following immediately holds.

**Claim 2.** *For any $X \in \mathcal{D} \setminus \{\mathsf{fStr}\}$, $\Pr[X = \mathsf{fStr}] = \frac{1}{2^\ell}$.*

This disposes off the cases of $\mathsf{fStr}$ colliding with any variable in $\mathcal{D}$. From the second point mentioned above, the following result is easily obtained.

**Claim 3.** *1. If one of $ty^s$ or $ty^t$ equals dec and $s \neq t$, then $\Pr[F_1^s = F_1^t] = 1/2^\ell$.*
*2. If one of $ty^s$ or $ty^t$ equals enc and $s \neq t$, then $\Pr[F_2^s = F_2^t] = 1/2^\ell$.*
*3. If $ty^s = enc$ or $ty^t = dec$, then $\Pr[F_2^s = F_1^t] = 1/2^\ell$.*

By the above claims, we are left only with the following cases to settle.

1. $s \neq t$, $ty^s = ty^t = enc$ and possible collision between $F_1^s$ and $F_1^t$.
2. $s \neq t$, $ty^s = ty^t = dec$ and possible collision between $F_2^s$ and $F_2^t$.
3. $ty^s = enc$, $ty^t = dec$ and possible collision between $F_1^s$ and $F_2^t$.

These are settled by the following two claims.

**Claim 4.** *1. If $s \neq t$ and $ty^s = ty^t = enc$, then $\Pr[F_1^s = F_1^t] \leq \frac{1}{2^\ell}$.*
*2. If $s \neq t$ and $ty^s = ty^t = dec$, then $\Pr[F_2^s = F_2^t] \leq \frac{1}{2^\ell}$.*

*Proof.* We provide the details of only the first point, the second point being similar. Consider two encryption queries $(P^s, T^s)$ and $(P^t, T^t)$, where $P^s = P_1^s||P_2^s||P_3^s$ and $P^t = P_1^t||P_2^t||P_3^t$. Then $(P^s, T^s) \neq (P^t, T^t)$ as $\mathcal{A}$ is not allowed to repeat queries. Recall

$$A_1^s = P_1^s;\ A_2^s = P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T_3^s);\ F_1^s = A_2^s \oplus H_{\tau''}(A_1^s).$$

We compute as follows.

$$\Pr[F_1^s = F_1^t] = \Pr\left[P_2^s \oplus H_{\tau'}(P_3^s||T_3^s) \oplus H_{\tau''}(P_1^s) = P_2^t \oplus H_{\tau'}(P_3^t||T_3^t) \oplus H_{\tau''}(P_1^t)\right]. \tag{18}$$

There are three cases to consider.
**Case 1.** $P_1^s \neq P_1^t$: In this case, the XOR universality of $H$ keyed by $\tau''$ shows the result.
**Case 2.** $P_1^s = P_1^t$, $(P_3^s||T_3^s) \neq (P_3^t||T_3^t)$: In this case, the XOR universality of $H$ keyed by $\tau'$ shows the result.
**Case 3.** $P_1^s = P_1^t$, $(P_3^s||T_3^s) = (P_3^t||T_3^t)$: Since $(P^s, T^s) \neq (P^t, T^t)$, it must hold that $P_2^s \neq P_2^t$ and so in this case the probability in 18 is 0. □

**Claim 5.** *If $ty^s = enc$ and $ty^t = dec$, then $\Pr[F_1^s = F_2^t] \leq \frac{1}{2^{\ell-1}}$.*

*Proof.* We consider an encryption query $(P^s, T^s)$ and a decryption query $(C^t, T^t)$, where $P^s = P_1^s||P_2^s||P_3^s$ and $C^t = C_1^t||C_2^t||C_3^t$. As before, recall the following expressions for $F_1^s$ and $F_2^t$.

$$F_1^s = P_2^s \oplus \beta \oplus H_{\tau'}(P_3^s||T^s) \oplus H_{\tau''}(P_1^s),$$
$$F_2^t = C_1^t \oplus (\beta \lll 1) \oplus H_{\tau'}(C_3^t||T^s) \oplus H_{\tau''}(C_2^t).$$

Let $\mathsf{rest}_1 = P_2^s \oplus H_{\tau'}(P_3^s||T^s) \oplus H_{\tau''}(P_1^s)$ and $\mathsf{rest}_2 = C_1^t \oplus H_{\tau'}(C_3^t||T^s) \oplus H_{\tau''}(C_2^t)$. Note that $\beta$ is independent of $\mathsf{rest}_1$ and $\mathsf{rest}_2$. So,

$$\Pr[F_1^s = F_2^t] = \Pr[\beta \oplus (\beta \lll 1) = \mathsf{rest}_1 \oplus \mathsf{rest}_2] \le \frac{1}{2^{\ell-1}}.$$

The last inequality follows from Lemma 1. $\square$

Based on the Claims 2 to 5, we can conclude that for distinct $X, Y \in \mathcal{D} \setminus \{\mathsf{fStr}\}$, $\Pr[X = Y] \le 1/2^{\ell-1}$. There are a total of $2q+1$ elements in $\mathcal{D}$. The element $\mathsf{fStr}$ in $\mathcal{D}$ is equal to any of the other elements with probability $1/2^{\ell}$ and this contributes $2q/2^{\ell}$ to $\Pr[\mathsf{COLLD}]$. Combining this with the probability of the other kinds of collisions is done as follows.

$$\Pr[\mathsf{COLLD}] \le \frac{2q}{2^{\ell}} + \binom{2q}{2}\frac{1}{2^{\ell-1}} \le \frac{4q^2}{2^{\ell}}.$$

Using (11), (16), (17) and (19)

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{STES}[\delta]}(\mathcal{A}) \le \frac{4q^2}{2^{\ell}} + \binom{q}{2}\frac{1}{2^{\ell}} \le \frac{9q^2}{2^{\ell+1}}.$$

Since $\mathcal{A}$ is an arbitrary adversary making $q$ queries, the theorem follows. $\square$

## 6 Hardware Implementation of STES

STES can be instantiated in various ways by plugging in different possibilities for the stream cipher and the hash function. There are, however, some common design ideas. The goal of this section is to describe the basic ideas behind the different implementations. Since we have implemented a number of designs, it is not possible to describe the details of all the designs. Neither is this necessary. From our descriptions of certain specific designs, it is possible to understand the details of the other designs. Results, however, are presented for all the implementations and these are consistent with our design goals of low area/power implementations.

Below we describe our basic design decisions. Then comes the descriptions of individual implementations of the different hash functions and the stream ciphers followed by the description for the STES implementation and the data flow and timing analysis.

### 6.1 Basic Design Decisions

The important design decisions are the following.

**Message length:** Our target is encryption of fixed size blocks. So, STES has been designed for fixed length messages. In particular, we consider the message length to be 512 bytes. This value matches the current size of memory blocks. It is to be observed that the design philosophies are quite general and can be scaled suitably for other message lengths.

**Stream cipher:** We chose the following stream ciphers: Grain128 [28], Trivium [13] and Mickey128 2.0 [6]. These are the eStream [3] finalists of hardware oriented designs. There are several works in the literature which reports compact hardware implementations of these ciphers [11, 19, 22].

There are different ways to implement these ciphers with varying amount of hardware cost. In particular, Grain128 and Trivium is amenable to parallelization and one can adopt strategies to design hardware which can give an output of only one bit per clock as in [11] or exploit the parallelization and increase the data-path to give more throughput at the cost of more hardware as in [28] and [13]. For instantiations with Grain128 and Trivium we tried different data paths for the stream ciphers and thus implemented multiple designs which provide a wide range of throughput. As mentioned in [29], there exists no trivial way to parallelize Mickey. So, our implementations of Mickey uses a data path of one bit. The various parameters considered for implementing the stream ciphers are shown in Table 1.

**Hash function:** The main component required to implement MLUH of PD is a finite field multiplier. In Section 2.2, we describe MLUH parameterized on the data path $d$, which signifies that the multiplications in MLUH with data path $d$ take place in the field $GF(2^d)$. We consider values of $d$ equal to 4, 8, 16, 32 and 40. The corresponding irreducible polynomials used to perform field multiplications are given in Table 2. The number of multipliers used for implementing the MLUH varies with the value of the data path.

The case of PD is a bit curious. For implementing PD with data path $d$ we use multipliers in $GF(2^{\frac{d}{2}})$. This decision was taken to make the design suitable for STES. Note that in case of PD each multiplication requires two message and key blocks which is not the case in MLUH. As the message and key blocks are generated on the fly hence this design helps in preventing data stalls in the circuit. This issue is discussed in more details in Sec. 6.2.

**Target FPGA:** We target our designs for Xilinx Spartan 3 and Lattice ICE40 FPGAs. The rationale is that these are considered suitable for implementing hardware/power constrained designs. Moreover, they are cheap and one can consider deploying these FPGAs directly into a commercial device. In particular, in Spartan III the LUTs within SLICEM can be used as a $16 \times 1$-bit distributed RAM or as a 16-bit shift register (SRL16 primitive). This functionality of a 16-bit shift register has been previously exploited to achieve compact implementations of stream ciphers [11]. We follow this suggestion.

The ICE40 FPGAs does not provide such functionalities. On the other hand, their architectural design specifically supports low power implementations. Our experimental results also suggest that they are much more competitive in this respect compared to Spartan 3 devices. To the best of our knowledge, there are no previous work reporting stream cipher implementation on this class of FPGAs.

| **Field** | $\lvert IV\rvert$ **(bits)** | $\lvert K\rvert$ **(bits)** | **Data paths used (bits)** |
|---|---|---|---|
| Trivium | 80 | 80 | $1, 4, 8, 16, 40$ |
| Grain128 | 96 | 128 | $1, 4, 8, 16, 32$ |
| Mickey128-2.0 | 96 | 128 | 1 |

**Table 1.** Specific parameters used to implement Trivium, Grain128 and Mickey128 2.0.

| **Field** | **Irreducible Polynomial** | **Field** | **Irreducible Polynomial** |
|---|---|---|---|
| $GF(2^4)$ | $x^4+x+1$ | $GF(2^{20})$ | $x^{20}+x^3+1$ |
| $GF(2^8)$ | $x^8+x^4+x^3+x+1$ | $GF(2^{32})$ | $x^{32}+x^7+x^3+x+1$ |
| $GF(2^{16})$ | $x^{16}+x^5+x^3+x+1$ | $GF(2^{40})$ | $x^{40}+x^5+x^4+x^3+1$ |

**Table 2.** Irreducible Polynomials.

## 6.2 Implementation of the Universal Hash Function

Implementations of the two hash functions MLUH and PD are described separately. In both cases, the size of the digest is equal to $\ell$ which is the size of the IV of the stream cipher.

**Design of MLUH:** An MLUH with data path $d$ denotes that the multiplications are performed in $GF(2^d)$. We choose $d$ such that $d$ divides $\ell$ and as before, we denote $b = d/\ell$. For convenience of exposition, we recall the description of $\mathsf{MLUH}^d_K(M)$ from 1.

$$\left.\begin{array}{l} h_1 = M_1 \cdot K_1 \oplus M_2 \cdot K_2 \oplus ... \oplus M_m \cdot K_m \\ h_2 = M_1 \cdot K_2 \oplus M_2 \cdot K_3 \oplus ... \oplus M_m \cdot K_{m+1} \\ \quad . \; . \; . \, . \, . \\ h_b = M_1 \cdot K_b \oplus M_2 \cdot K_{b+1} \oplus .... \oplus M_m \cdot K_{b+m-1}. \end{array}\right\} \tag{19}$$

The basic strategy for the above computation is to apply $b$ different multipliers. For a fixed $\ell$, since $b = d/\ell$, as $d$ grows, the value of $b$ decreases. The computation proceeds column-wise. The $b$ multiplications $M_1 \cdot K_1, M_1 \cdot K_2, \ldots, M_1 \cdot K_b$ are performed first. Since there are $b$ multipliers, these can be done in parallel. The results of these multiplications are stored separately. In the next step, the products $M_2 \cdot K_2, M_2 \cdot K_3, \ldots, M_2 \cdot K_{b+1}$ are again performed in parallel and these results are xor-ed with the previous results. This is continued until all columns have been computed.

In Figure 6, we showcase the above strategy of computing MLUH with a specific architecture for $d = 8$ and $\ell = 80$, so that $b = 10$, i.e., there are 10 multipliers where each multiplier can multiply two elements of $GF(2^8)$. The whole architecture consists of ten 8-bit registers, ten multipliers and ten 8-bit accumulators. All the registers are connected in cascade forming a 10-stage first in first out (FIFO) structure with parallel access to all states. In Figure 6, the registers are labeled as **regk1**, **regk2**, ..., **regk10**. These registers are used to store ten 8-bit blocks of the key. Each multiplier takes one of its inputs from the FIFO structure and the other takes its input directly from the input line $m_i$. Initially all registers in the FIFO and accumulators have zero value.

The FIFO is fed with the key blocks $K_1, K_2, \ldots$, etcetera, one in each cycle through the input line depicted $k_i$ in the figure. After ten clock cycles, the FIFO is full, i.e, the registers contain the key blocks $K_1, K_2, \ldots, K_{10}$ and the input line $m_i$ contains the message block $M_1$ and the multiplications in the first column of MLUH are performed. Then each product is accumulated in the respective accumulators. In the next clock, the FIFO contains the key blocks $K_2, \ldots K_{11}$ and the input line $m_i$ contains the message block $M_2$; the second column of multiplications are computed and these results are accumulated in the respective registers. This is continued until all the columns have been processed. The final output of MULH is obtained by concatenating the final values in the accumulators. The control unit is not shown in the figure, it consist of a counter and some comparators.

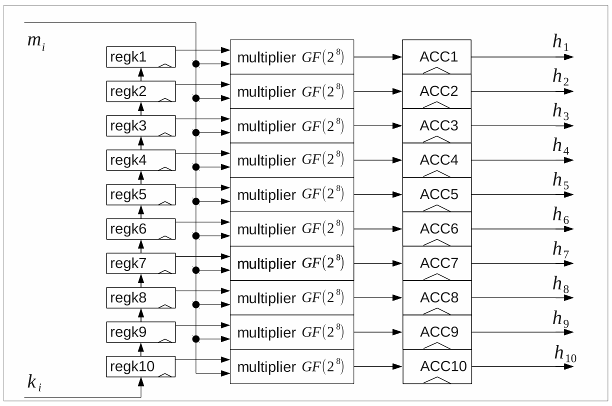

**Fig. 6.** Architecture of MLUH

**Design of PD:** For hashing a $m$-block message to a $b$ block output, where each block in $d$ bit long, MLUH requires $mb$ multiplications in $GF(2^d)$. In comparison, PD requires only $\frac{mb}{2}$ multiplications (assuming $m$ is even). So, the total number of multiplications required by PD is about half that of MLUH. This suggests that PD should be the design of choice. But, that is not necessarily true, at least in our context, as we describe below.

Each multiplication in PD is of the form $(M_i \oplus K_j)(M_{i+1} \oplus K_{j+1})$. So, performing this multiplication requires two blocks of message and key material. In contrast, for MLUH a multiplication can be performed when one block of message and key are available. This difference is important.

In STES, PD is to be used in conjunction with a stream cipher. For the second hashing in the encryption algorithm, the message and key blocks to be used by PD are obtained as an output from the stream cipher. To keep this balance, we decided to construct a PD with $d/2$ bit multipliers when the message and key blocks are considered to be $d$ bit blocks. Here we showcase an architecture for PD with 4 bit multipliers which produces a 80 bit output. Later, we use this PD construction with a stream cipher with a 8 bit data path to construct STES.

We implement the PD as is shown in Figure 7. The methodology adopted is the same as in case of the architecture of MLUH (shown in Figure 6), in the sense that here also we compute column wise. But as we use 4 bit multipliers, to get a 80 bit output we require 20 multipliers. We assume that the key blocks ($K_j$) and message blocks ($M_i$) are obtained as 8 bit blocks, but we treat each multiplication as $(M_i^H \oplus K_j^H)(M_i^L \oplus K_j^L)$, where $M_i = M_i^H||M_i^L$ and $K_j = K_j^H||K_j^L$ and $|M_i^H| = |M_i^L| = |K_j^H| = |K_j^L| = 4$. Here we have twenty registers named $\mathbf{regK1}, \mathbf{regK2}, \ldots, \mathbf{regK20}$, each of length 8 bits connected in a cascade forming a FIFO as in case of MLUH architecture. These registers contains the key blocks, and the message blocks are obtained from the input lines $m_i^H$ and $m_i^L$.

This 4 bit design of the PD is not more efficient than the MLUH architecture. Though the 4 bit multipliers used in PD are smaller than the 8 bit multipliers but we require the double of them and also the double amount of registers and the extra xors required at the input of the multipliers makes this PD architecture marginally more costly than the MLUH architecture. Moreover the number of clock cycles required in this case is also same as in case of MLUH.

Our experiments (presented later) shows PD to be more efficient in terms of area for higher data paths. This can be explained by an intuitive argument. The asymptotic area complexity of a $k$ bit multiplier is super-linear in $k$, thus the total area of a $k$ bit multiplier is expected to be larger than two $k/2$-bit multipliers for large values of $k$.

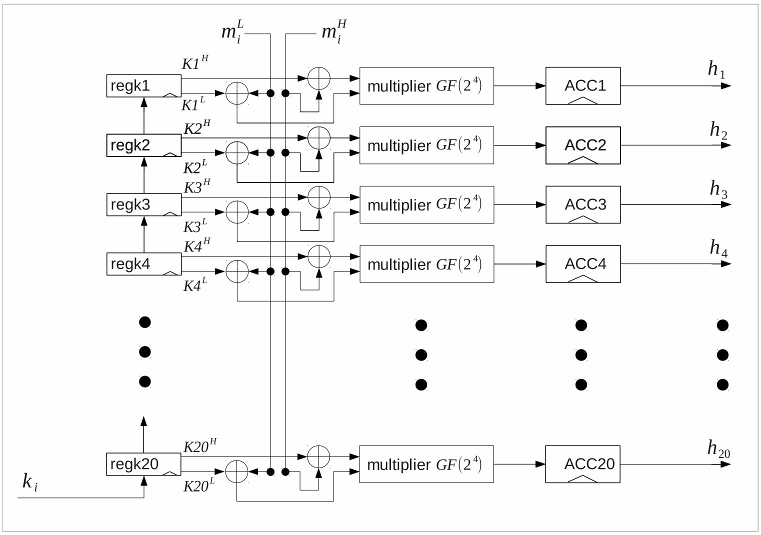

**Fig. 7.** Architecture of PD.

The multipliers used in both PD and MLUH are Karatsuba multipliers, they were implemented following the same design strategy as presented in [15]. The irreducible polynomials used to implement the multipliers are listed in the Table 2, also these multipliers are smaller than the ones presented in [15] as they operates on smaller numbers. To keep the speed high and seeing that there are no dependencies between multiplications in MLUH and PD, after a careful re-timing process the multipliers for $d > 4$ were divided into almost balanced pipeline stages, the specific number of stages used in each implementation is reported in Tables 3 and 4.

## 6.3 Implementation of stream ciphers

In this work we consider three stream ciphers: Trivium, Grain128 and Mickey128-2.0. These stream ciphers are the eStream hardware based stream ciphers and are in general very easy to implement in hardware as they are constructed using simple structures like shift registers and some simple Boolean functions. All these three stream ciphers can be implemented using a shift register as a basic primitive.

We implement the stream cipher using various data paths, here by a data path we mean the number of bits of output the stream cipher can produce in each clock cycle. A lower data path uses less parallelism and thus can be implemented with fewer hardware resources. The various data paths that we consider for the three stream ciphers along with some other important parameters are depicted in Table 1.

For all the three stream ciphers, the bit-wise versions (i.e. the ones with data path of one bit) can be implemented in a very compact way in Spartan-3 devices. Spartan-3 FPGAs can configure the Look-Up Table (LUT) in a SLICEM slice as a 16-bit shift register without using the flip-flops available in each slice. Shift-in operations are synchronous with the clock, and output length is dynamically selectable. A separate dedicated output allows the cascading of any number of 16-bit shift registers to create whatever size shift register is needed. Each configurable logic block can be configured using four of the eight LUTs as a 64-bit shift register. Such an usage of the LUT in Spartan-3 is called a SRL16 primitive [43]. This SRL16 primitive can be used to implement the shift registers of the stream ciphers [11]. SRL16 supports only a single entry and a single bit shift, so if the data path is more than 1 then this primitive cannot be used and then the shift registers must be implemented using simple LUTs. We implemented bit-wise versions of Trivium and Grain128 using SRL16 primitive. A bit-wise version of Mickey128 2.0 was also implemented, but the structure of Mickey128 2.0 does not allow efficient use of SRL16. Also, we did not implement Mickey128-2.0 with data paths more than 1, as such parallelization in Mickey is not straight forward to obtain.

Here, as an example we will explain in details the a specific architecture of Trivium with a 2-bit datapath. The internal state of Trivium is a 288-bit shift register, for implementation purposes it is divided into a three registers $SR1$, $SR2$ and $SR3$ as shown in Figure 8. All the three shift registers have two inputs and two outputs and in each clock cycle their internal states are shifted by two positions. Initially $SR1$ and $SR2$ are initialized with the 80 bit key $K$ and the 80 bit $IV$ respectively. $SR3$ has as initial value the string $1^3||0^{108}$. In the Figure 8 it can be seen that for each shift register its feedback functions depends on some bits from it and a function computed with some bits from the previous register. For example, the feedback of shift register $SR3$ depends on some bits of $SR2$ and two bits from itself. It is easy to see in Figure 8 that the feedback functions for all registers and the function to compute the final outputs $S_{even}$ and $S_{odd}$ are replicated two times, just they have different inputs. In the case of Gran128 the way to increment the datapath also consist of replicating the feedback functions of shift registers and the output function. Increasing the datapath of Grain128 and Trivum brings a significant increase in throughput since it reduces the time used for setup and give a parallel output for the stream.

## 6.4 Implementation of STES

We implemented STES with all the three stream ciphers with the data-paths specified in Table 1. When we consider a stream cipher with data path $d$ in implementing STES then we use the hash function with the same value of the data path, i.e., we use multipliers in $GF(2^d)$ if the hash function is MLUH, and if it is PD then the multipliers are in $GF(2^{\frac{d}{2}})$.

We will explain in details a 8-bit data path implementation using Trivium and MLUH, but for other instantiations of stream ciphers and hash the basic design remains the same. Note that Trivium uses a 80 bit $IV$ and a 80 bit key.

In the Figure 9 we show the generic architecture for encrypting/decrypting with STES, we shall explain the architecture with reference to the algorithm of STES (Figure 1) and the Feistel network (Figure 2).

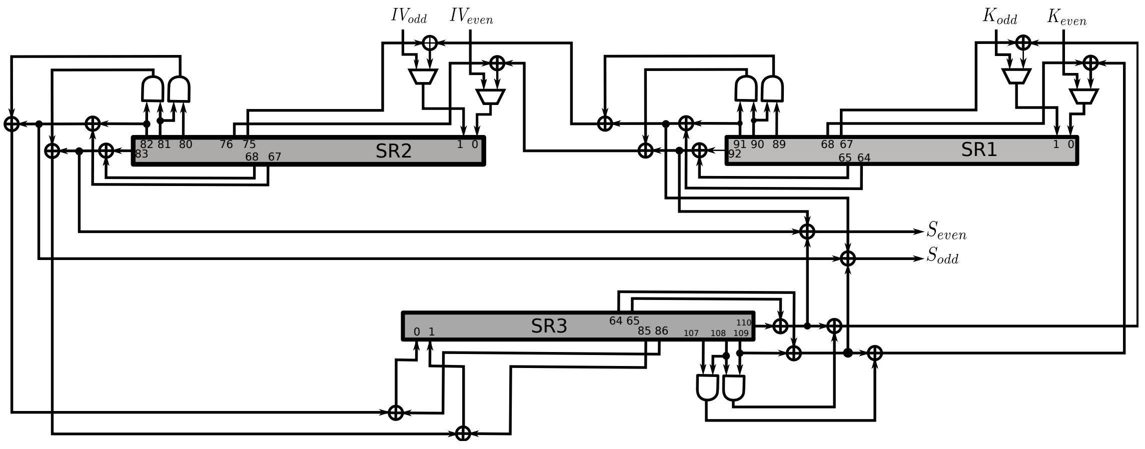

**Fig. 8.** Architecture for Trivium.

The circuit presented in Figure 9 consists of the following basic elements:

1. The MLUH constructed with 8 bit multipliers as discussed in Section 2.2. In the diagram this component is labeled **MLUH**.
2. Two stream cipher cores labeled **SC1** and **SC2**.
3. Two 80 bit registers **RegH1** and **RegH2** which are used to store the output of **MLUH**.
4. Four registers labeled **regF1**, **regF2**, **regKh** and **reg**$\beta$. All these registers are 80 bit long and are formed by ten registers each of eight bits connected in cascade, so that they can be used as a FIFO queue. The same structure was used in the design of MLUH and PD.
5. One special register **reg**$\beta$**1** which is able to store a 80-bit data and rotate it in one bit position. This register outputs 8-bit data each clock cycle when the control input $ce$ is activated.
6. Seven multiplexers labeled **1**, **2**, **3**, **4**, **5**, **6** and **7**.
7. The control unit whose details are not shown in the Figure.
8. The connections between **MLUH** and the registers **RegH1**, **RegH2** have a data path of 80 bits. All other connections have a data path of 8 bits.
9. The input lines $M_i$, $IV$ and $K$ which receives the data and tweak, the initialization vector and the key respectively.
10. The output line $C_i$ which outputs the cipher.

The **MLUH** computes the MLUH, it receives as inputs message blocks $M_i$, tweak blocks $T_i$ and key blocks $K_i$ and give as output the result of MLUH in its output port S. The register **RegH1** and **RegH2** receive the output from S as input, in this case $|S| = 80$ bits. The registers **RegH1** and **RegH2** are designed to give eight bit blocks as outputs in each clock cycle in their output port BO. The **MLUH** receives its input from the $3 \times 1$ multiplexer labeled **1**. Notice, that in the algorithm of STES, the MLUH is called on three different inputs. Multiplexer **1** helps in selecting these inputs. In the algorithm MLUH is called on two different keys $\tau'$ and $\tau''$, thus, **MLUH** can receive the key from two different sources: the key $\tau'$ is received directly from the output of the stream cipher **SC1** or **SC2**. The key $\tau''$ is received either directly from stream cipher **SC1** or from the register **regKh** which is used to store $\tau''$. To accommodate these selection of keys the input port Ki of **MLUH** receives the input from the $2 \times 1$ multiplexer **5**.

We use two stream ciphers **SC1** and **SC2**. Both take the key from the input line $K$ of the circuit. **SC1** receives the IV from multiplexer **2**, it selects between input line $IV$ or $F_1$. Multiplexer **3** feeds the IV to the stream cipher **SC2**, it selects between $IV$ or $F_2$.

In the algorithm of STES we can see that the output of MLUH is xored with the value of $\beta$ or $\beta \lll 1$ depending which hash is computed $Z_1$ or $Z_2$ and whether encryption or decryption mode is being executing. The selection between these two values is made with Multiplexer **7**.

In the encryption mode the stream $W$ is generated using **SC2** but in the decryption mode it is generated by **SC1**. Multiplexer **6** is used to select the correct stream cipher to produce the cipher text or plain text.

Next we explain the data-flow of the architecture of the Figure 9 with reference to the algorithm in Figure 1.

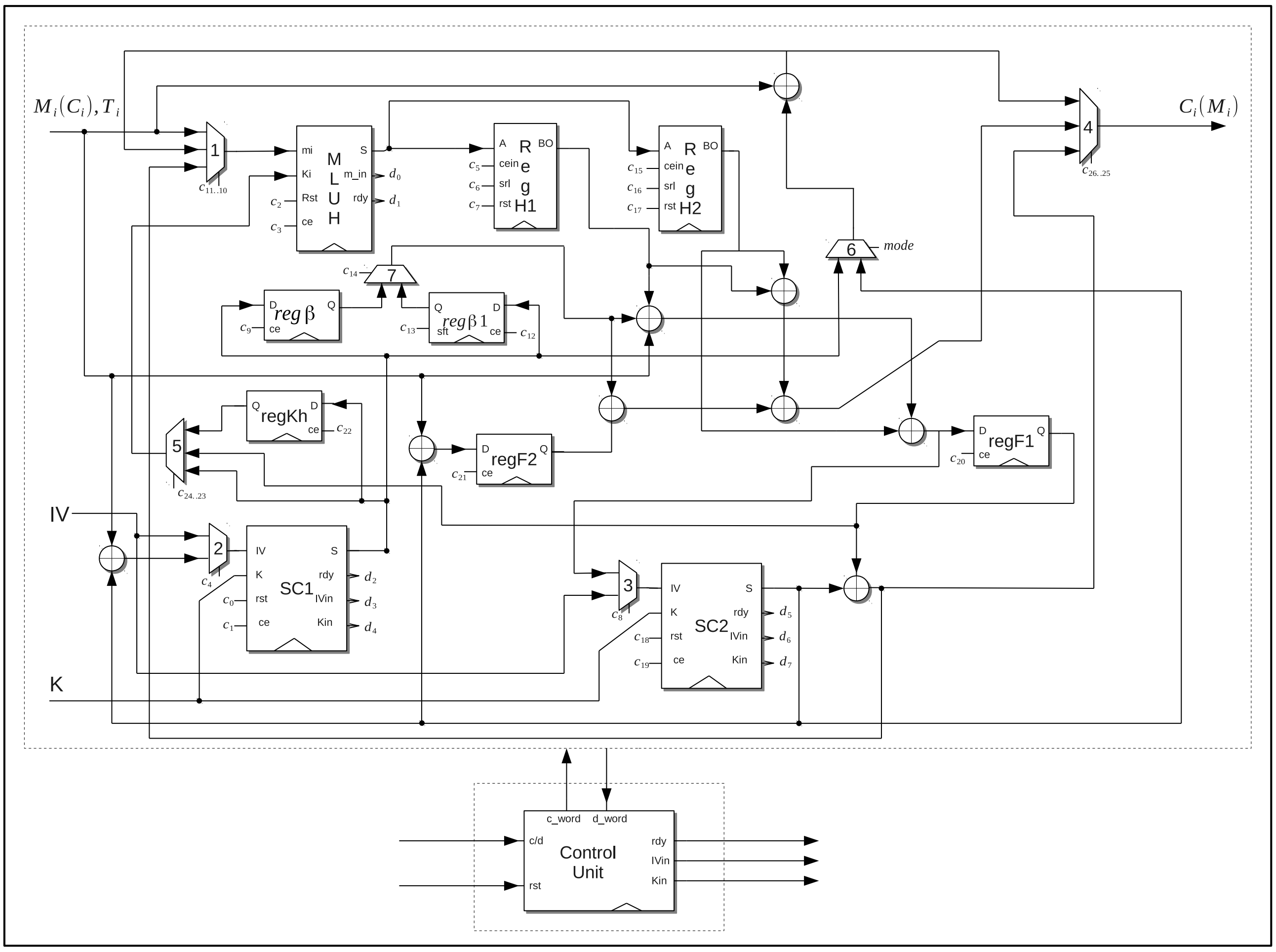

**Fig. 9.** Architecture of complete STES

### 6.5 Data Flow and Timing Analysis

In this section we shall discuss the data flow in the circuit presented in Figure 9 and also discuss the parallelism that we achieve using the circuit and the latency of the various operations, we present a time diagram depicting the time taken by each operation in terms of clock cycles in Figure 10. In Figure 10 the basic operations are depicted with rectangles and the numbers inside the rectangles denotes the time required for the operation in clock cycles.

We consider encryption of a message $P$ of length 4096 bits. As per the algorithm $P$ is parsed into three blocks $P_1$, $P_2$, $P_3$, where $|P_1| = |P_2| = \ell = 80$ and $|P_3||T| = 4016$. The encryption procedure starts with the computation of the key bits $\tau = \tau'||\beta||\tau''$ using the stream cipher **SC1**. It is required to generate 540 bytes of key material using the stream cipher. Our implementation for Trivium takes 154 cycles for key setup, and to generate 540 bytes it again takes 540 cycles. The key bytes generated by **SC1** is parsed as $\tau'$, $\beta$ and $\tau''$. The key bytes $\tau'$ are not stored and are immediately fed to the **MLUH** as it gets generated. Note, that these key bytes are used again and then they are again generated using SC1 or SC2. Storage of this huge key material would amount to more area of the circuit in terms of extra registers hence we decided against the option of storing it. $\beta$ is stored in **reg**$\beta$ and $\beta \lll 1$ in **reg**$\beta$**1**.

$\tau''$ is used twice inside the Feistel network, and the size of $\tau''$ is much smaller than $\tau'$, hence it is stored in **regKh**.

As soon as the key set up phase of **SC1** is over, in each clock it generates one byte of key material and these key materials gets stored in the FIFO register of MLUH one byte per cycle. After 10 cycles the FIFO is full and thus the MLUH can start computing $Z_1$. It takes 513 cycles to complete the computation of MLUH and it runs in parallel with the stream cipher, note that to finish to compute $Z_1$ we need $\beta$, to generate $\beta$ 10 clock cycles more are necessaries therefore in Figure 10 the computation of $Z_1$ takes 523 clock cycles. We store $Z_1 \oplus \beta$ (i.e., the MLUH part of $Z_1$) in **RegH1**.

Once $\beta$ has been computed the stream cipher **SC1** starts generating the $\tau''$ part of the key. $\tau''$ and $A_1 = P_1$ are fed to **MLUH** to compute $H_1$ (line 1 of Fig. 2), $H_1$ is stored in the register **RegH2**. Using the value of $H_1$ in **RegH2** and the values in **RegH1** and **reg**$\beta$ $F_1$ is computed, and stored in **regF1**. The value of $F_1$ is then fed to **SC2** as an initialization vector to compute $G_1$. After the initialization of **SC2** is completed and it starts producing $G_1$, then computation of $F_2$ is started and stored in **RegF2**, in parallel it is fed to **SC1** for computation of $G_2$. After $G_1$ has been produced, **SC2** is inactivated using the control input *ce*. This inactivation is done as the following bits of the stream correspond to $W$, which is produced later. When the initialization process of **SC1** ends, $G_2$ starts getting computed and it is used with the value stored in **regF1** to compute $B_2$ which is given as ciphertext output $C_2$. Also $B_2$ is fed to the **MLUH** to compute $H_2$ using the key stored in **regKh**. When $H_2$ is ready it is stored in **RegH2**. This almost completes the evaluation of the Feistel in line 16 of the algorithm in Fig. 1, the computation up to this stage is shown as Fiestel in Fig. 10 and it takes 350 cycles. For convenience $B_2$ is computed along with $C_1$ later. After the Fiestel part is completed, **SC1** generates the key $\tau'$ for the MLUH computation in line 18, after the initialization of **SC1**, **SC2** is activated to generate $W$. Using $W$ and input $P_3$ the output $C_3||T$ is computed and fed to **MLUH**. When all $C_3||T$ are processed by **MLUH** the result is stored in **RegH1**.

Finally output $C_1$ is computed using the values stored in **RegH1**, **RegH2**, **reg$\beta$1** and **RegF2**. The total time taken to encrypt 512 bytes is 1705 cycles.

**Decryption process:** There are some differences in the data flow when the decryption process is performed. The most significant is inside the Feistel network. For decryption $W$ is computed with **SC1** hence **SC2** can be used to compute the key $\tau'$. The initialization process of **SC2** is realized in parallel with a part of initialization of **SC1** and the computation of $H_2$, this represent a saving of one stream cipher initialization which is 154 clock cycles in case of Trivium. This difference is depicted in Figure 10 with the dashed box in $\tau'$ computation. Thus, for our circuit the process takes 1551 clock cycles.

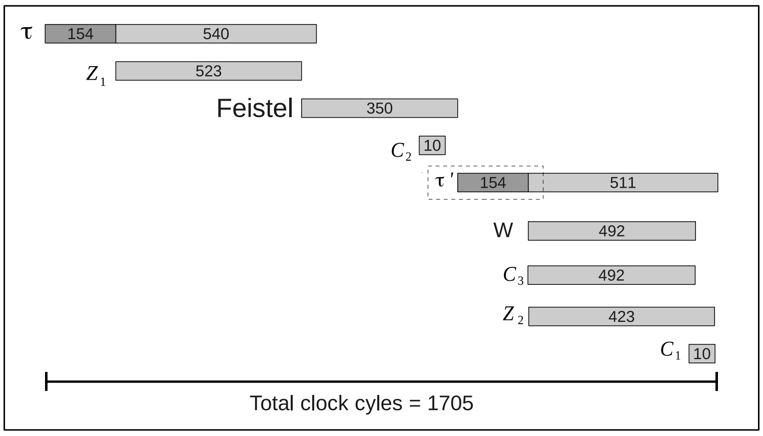

**Fig. 10.** Time diagram for encrypting 512 bytes with STES. For decryption the timings are similar, but only the boxed part is not required.

## 7 Experimental Results

We implemented STES on two different families of FPGAs: Lattice ICE40 and Xilinx Spartan 3. For Spartan 3 we used the device *xc3s400-fg456* and in case of Lattice ICE40 we selected *LP8KCM225*. The place and route results in case of Spartan 3 were generated using *Xilinx-ISE* version 10.1. For ICE40 we used Silicon Blue Tech iCEcube release 2011.12.19577. We measured the power consumption of the circuits using *Xilinx Xpower Analyzer* for Spartan 3 and Power Estimator of iCEcube for ICE40.

For our implementations we report performance in terms of throughput, area and power-consumption. In case of Spartan 3 we report area in terms of number of slices and for ICE40 we report in terms of number of logic cells. It is to be noted that size of a Spartan 3 slice is almost equal to twice the size of a ICE40 logic cell.

In this section we present the experimental results in two parts. First in Section 7.1 we report performance data of the primitives, i.e., the stream ciphers and the hash function and in Section 7.2 we report results of STES using various instantiations of the primitives.

## 7.1 Primitives

In the Tables 3 and 4 we show the performance results of MLUH and PD implementations. For all the cases, we consider hashing a message of 4016 bits to 80 bits. In the Tables MLUH-$d$b represents an MLUH with a $d$ bit data path. And PD-$db$ represents a PD construction with $d$ bit data path and multipliers in $GF(2^{\frac{d}{2}})$(see the architecture discussed in Section 6.2).

It is clear from the Tables that in both Spartan 3 and ICE40 with the increase in data path the throughput increases at the cost of area. For MLUH-1b, we obtain a very high frequency, as in this case the multiplier is only an AND gate. As the data path increases, the complexity of the circuit implementing the multiplication grows which increases the critical path of the circuit. For 16, 32 and 40 bit implementations we break the critical path of the multiplier by dividing it into balanced pipeline stages, the number of pipeline stages were carefully selected to maintain a high operating frequency. This is the reason why all 8, 16, 32 and 40 bit implementations operate on similar frequencies on both Spartan 3 and ICE40.

In Table 4 the performance of PD is shown. For 4 bit data paths, the size of PD is a bit bigger than MLUH but for bigger data paths the size of PD is smaller. This conforms to the argument that we provided in Section 6.2. The number of cycles required to compute PD is same as of MLUH, but due to the more complex circuitry of PD it operates in a bit lower frequencies and thus achieves lesser throughput than MLUH.

| | SPARTAN 3 | | | | LATTICE ICE40 | | | |
|---|---|---|---|---|---|---|---|---|
| **Primitive** | **Slices** | **Freq. (MHz)** | **Pipeline Stages** | **Thrghpt (Mbps)** | **Logic Cells** | **Freq. (MHz)** | **Pipeline Stages** | **Thrghpt (Mbps)** |
| MLUH-1b | 158 | 215.11 | 0 | 210.90 | 325 | 189.78 | 0 | 189.78 |
| MLUH-4b | 247 | 183.76 | 0 | 720.68 | 419 | 179.50 | 0 | 703.97 |
| MLUH-8b | 452 | 177.46 | 1 | 1389.24 | 772 | 172.89 | 1 | 1353.46 |
| MLUH-16b | 737 | 175.24 | 2 | 2773.07 | 1638 | 170.52 | 2 | 2644.05 |
| MLUH-40b | 1410 | 173.89 | 3 | 6625.64 | 2756 | 174.80 | 3 | 6622.31 |

**Table 3.** Performance of MLUH on Spartan 3 and Lattice ICE40.

| | SPARTAN 3 | | | | LATTICE ICE40 | | | |
|---|---|---|---|---|---|---|---|---|
| **Primitive** | **Slices** | **Freq. (MHz)** | **Pipeline Stages** | **Thrghpt (Mbps)** | **Logic Cells** | **Freq. (MHz)** | **Pipeline Stages** | **Thrghpt (Mbps)** |
| PD-4b | 266 | 172.65 | 0 | 690.60 | 490 | 170.68 | 0 | 669.38 |
| PD-8b | 364 | 167.50 | 0 | 1340.00 | 748 | 161.49 | 0 | 1266.68 |
| PD-16b | 710 | 169.87 | 1 | 2707.34 | 1023 | 165.71 | 1 | 2589.46 |
| PD-40b | 922 | 172.18 | 2 | 6623.32 | 1832 | 159.29 | 2 | 6127.48 |

**Table 4.** Performance of PD on Spartan 3 and Lattice ICE40.

In case of both MLUH and PD we can see that the number of logic cells required for ICE40 FPGA is almost double than the slices required in Spartan 3. It is to be noted that a logic cell in ICE40 has much lesser components than in a Spartan 3 slice, which explains the difference in area in the two families. Moreover, the ICE40 implementations operate at a little lower frequencies compared to the Spartan 3 implementations, this can also be explained by the fact that as a ICE40 has lesser components so the critical path of the implementations in ICE40 are more complex in terms of logic resources.

In Table 5 we present the performance data of Trivium, Grain and Mickey with various data paths. In the Tables the names of the stream ciphers are suffixed with the data path.

The bit-wise implementation of Trivium and Grain128 on Spartan 3 were done using SRL16 primitives and this allowed us to obtain very compact designs: 49 Slices for Trivium-1b and 67 Slices for Grain128-1b. Grain128-1b is larger than Trivium-1b due to the complexity of its output and feedback functions. Mickey128 was implemented only with a one bit data path because there is no direct way to parallelize it.

| | SPARTAN 3 | | | | LATTICE ICE40 | | | |
|---|---|---|---|---|---|---|---|---|
| **Primitive** | **Slices** | **Freq. (MHz)** | **Setup cycles** | **Thrghpt. (Mbps)** | **Logic Cells** | **Freq. (MHz)** | **Setup cycles** | **Thrghpt. (Mbps)** |
| Trivium-1b | 49 | 201.02 | 1232 | 201.02 | 313 | 190.26 | 1232 | 190.26 |
| Trivium-4b | 120 | 197.38 | 308 | 789.52 | 329 | 188.54 | 308 | 754.15 |
| Trivium-8b | 148 | 193.49 | 154 | 1547.93 | 347 | 186.13 | 154 | 1489.04 |
| Trivium-16b | 203 | 189.32 | 77 | 3029.16 | 398 | 176.10 | 77 | 217.60 |
| Trivium-40b | 278 | 187.83 | 31 | 6010.60 | 530 | 166.73 | 31 | 6669.20 |
| Grain128-1b | 67 | 193.00 | 384 | 193.00 | 297 | 192.59 | 384 | 192.58 |
| Grain128-4b | 175 | 182.54 | 96 | 730.17 | 360 | 162.41 | 96 | 649.64 |
| Grain128-8b | 232 | 178.80 | 48 | 1430.42 | 434 | 145.73 | 48 | 1165.84 |
| Grain128-16b | 320 | 179.73 | 24 | 2875.64 | 592 | 137.72 | 24 | 2203.53 |
| Grain128-32b | 490 | 173.58 | 6 | 5554.56 | 997 | 136.84 | 12 | 4378.88 |
| Mickey128-1b | 182 | 202.80 | 286 | 202.80 | 420 | 169.74 | 288 | 169.74 |

**Table 5.** Performance of the stream ciphers in Spartan3 and Lattice ICE40.

The data in Tables 5 shows that the increase in data path does not have much effect on the total area of Grain128 and Trivium. For example, Trivium-8b requires 148 slices and Trivium-16b requires 203 slices. Though one would expect that doubling the data path would require double the hardware resources, that is not the case. The growth in area is small because in Trivium the state is stored in a 288-bit shift register independent of the size of data-path. For wider data paths we only require to replicate the output and the feedback functions a suitable number of times.

Wider data path implementations of Grain128 also have the same behavior as implementations of Trivium. As Grain128 has a 96 bit IV hence for our requirement that the data path must divide the IV length we do not implement grain with a 40 bit data path which we do for Trivium.

## 7.2 Experimental results on STES

Using the primitives described in Section 7.1 we construct STES. The performance results are shown in Tables 6 and 7. The Tables shows data for STES implemented with various stream cipher instantiations and data paths. The Tables also show the power consumption characteristics for the implementations. Note that we did not include PD implementations for data paths less than 4 bits.

As our specific design of PD would not allow 1 bit data paths and for 2 bit data paths there is no advantage of PD over MLUH.

From Table 6 we can observe the following:

– Among the one bit data path implementations, STES with Trivium achieves the smallest area and STES with Mickey is the fastest closely followed by STES with Grain128. STES[Grain,MLUH]-1b has the best throughput per area metric.
– The implementations which use Grain128 are in general faster than the ones using Trivium, because the implementations with Grain needs less clock cycles in comparison with implementations with Trivium.
– The constructions with PD have lesser area than the constructions with MLUH for higher data paths. For 4 bit data path implementations the MLUH based construction is smaller, this is possibly due to the fact that the difference of area between a 2 bit multiplier and a 4 bit multiplier is not much.
– The constructions with PD operates at a slightly lower frequency than the MLUH based constructions, this is due to the fact that PD has a more complex circuitry. But both PD and MLUH based constructions takes the same number of clock cycles, thus the PD based constructions have a lower throughput.
– In Figures 11 and 12 we present the data in Tables 6 and 7 for STES instantiated with Trivium and Grain128 using MLUH in a pictorial form. Figure 11 shows the growth of area, throughput and total power for STES using Trivium and Figure 12 shows the same for STES using Grain. Note that the plots are in logarithmic scale. We can observe that with increase in data path the growth of throughput is much faster than the growth of area, this reflects the characteristic of Trivium as shown in Table 5. The growth of power consumption is the slowest.

In the Table 7 we present the experimental results for implementations of STES on ICE40. The comparative behavior reflected in the Table 7 is almost the same as the behavior of implementation on Spartan 3 shown in Table 6. One difference that we observe is that for the 8bit implementations STES[PD] is slightly larger in size than STES[ML] for both Grain and Trivium. In general the implementation on ICE40 are slower than the implementations on Spartan 3, but the power consumption on ICE20 is much better. In particular we observed that the static power consumption in ICE40 remains constant for all variants. This is probably due to the fact that ICE40 was specifically designed to be used in low power applications, hence its architecture has special characteristics which allows it to run with a very low power consumption.

As the implementations on ICE40 performs the best in terms of power consumption, hence we measured the performance of all our designs when the operating frequency was fixed to 100MHz. The ICECube software allows such simulations. In the Table 8 we show the power consumption of all implementations in ICE40 when operating at a fixed frequency of 100 Mhz. As expected, if we lower the operating frequency the circuits consume considerably lesser amount of power but at the cost of lower throughput.

## 7.3 Comparison with Block Cipher Based Constructions

As mentioned earlier STES is highly motivated by the construction presented in [39], here we present some results and estimations on the construction in [39].

| **Mode** | **Area** (Slices) | **Freq.** (MHz) | **Cycles** | | **Throughput** (Mbps) | | **TPA** | | **Static power (mW)** | **Dynamic power (mW)** | **Total power (mW)** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | enc | dec | enc | dec | enc | dec | | | |
| STES[Trv,ML]-1b | 528 | 160.15 | 13601 | 12369 | 48.18 | 52.98 | 22.30 | 24.52 | 54 | 51 | 105 |
| STES[Trv,ML]-4b | 683 | 159.16 | 3401 | 3093 | 192.03 | 211.14 | 68.71 | 75.55 | 60 | 57 | 117 |
| STES[Trv,PD]-4b | 714 | 158.56 | 3401 | 3093 | 190.76 | 209.77 | 65.30 | 71.80 | 61 | 70 | 131 |
| STES[Trv,ML]-8b | 1050 | 160.26 | 1705 | 1551 | 384.62 | 422.81 | 89.52 | 98.41 | 61 | 94 | 155 |
| STES[Trv,PD]-8b | 964 | 155.14 | 1705 | 1551 | 372.34 | 409.30 | 94.39 | 103.76 | 61 | 93 | 154 |
| STES[Trv,ML]-16b | 1346 | 158.89 | 859 | 782 | 756.90 | 831.43 | 137.52 | 150.95 | 62 | 191 | 253 |
| STES[Trv,PD]-16b | 1249 | 154.70 | 859 | 782 | 736.94 | 809.51 | 144.19 | 158.39 | 62 | 196 | 258 |
| STES[Trv,ML]-40b | 2433 | 154.34 | 353 | 323 | 1789.12 | 1960.14 | 179.71 | 196.88 | 102 | 256 | 319 |
| STES[Trv,PD]-40b | 2150 | 153.65 | 353 | 323 | 1781.12 | 1951.38 | 202.19 | 221.80 | 84 | 240 | 324 |
| STES[Grn,ML]-1b | 591 | 159.61 | 10337 | 9953 | 63.18 | 65.61 | 26.13 | 27.13 | 58 | 32 | 90 |
| STES[Grn,ML]-4b | 834 | 159.26 | 2585 | 2489 | 252.11 | 261.83 | 73.87 | 76.72 | 60 | 41 | 101 |
| STES[Grn,PD]-4b | 876 | 154.11 | 2585 | 2489 | 243.95 | 253.36 | 68.06 | 70.68 | 60 | 36 | 96 |
| STES[Grn,ML]-8b | 1200 | 159.79 | 1297 | 1249 | 504.13 | 523.50 | 102.67 | 106.61 | 61 | 108 | 169 |
| STES[Grn,PD]-8b | 1136 | 153.87 | 1297 | 1249 | 485.45 | 504.11 | 104.43 | 108.44 | 60 | 46 | 106 |
| STES[Grn,ML]-16b | 1657 | 158.78 | 655 | 631 | 991.950 | 1029.67 | 146.30 | 151.86 | 62 | 220 | 282 |
| STES[Grn,PD]-16b | 1598 | 153.90 | 655 | 631 | 961.46 | 998.03 | 147.04 | 152.63 | 62 | 212 | 274 |
| STES[Grn,ML]-32b | 2729 | 155.82 | 332 | 320 | 1920.57 | 1960.15 | 171.98 | 178.43 | 64 | 360 | 414 |
| STES[Grn,PD]-32b | 2595 | 149.50 | 332 | 320 | 1842.63 | 1911.73 | 173.53 | 180.03 | 63 | 256 | 319 |
| STES[Mky,ML]-1b | 835 | 152.92 | 9953 | 9665 | 62.87 | 64.74 | 18.40 | 18.95 | 70 | 52 | 132 |

**Table 6.** Performance of STES on Spartan 3. STES[SC,Hash]-$d$b denotes STES of data path $d$ instantiated with stream cipher SC and hash function Hash. Trv: Trivium, Grn: Grain128, Mky: Mickey128. ML: Multilinear universal hash, PD: Pseudo Dot Product. TPA : Throughput per area, is computed as $\frac{1}{\text{area}\times\text{time}}$, where area is in slices and time in microseconds for encrypting/decrypting 512 bytes.

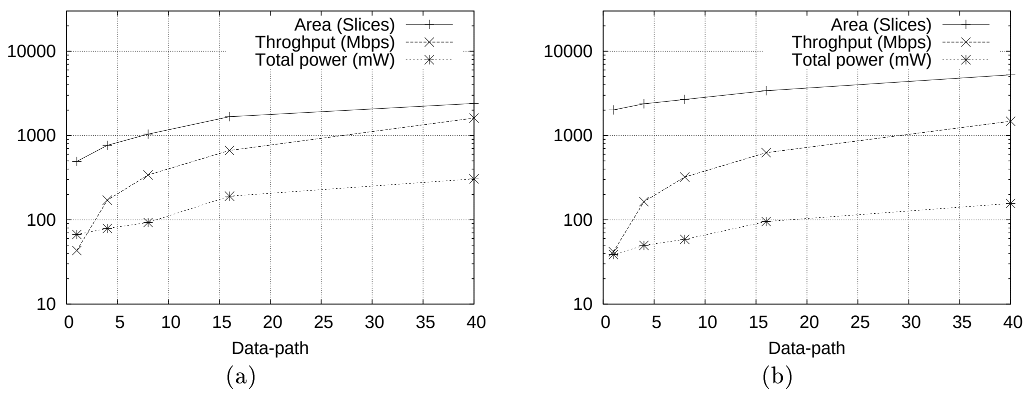

**Fig. 11.** Growth of Area, Throughput and Total power for STES[Trivium,MLUH]: (a) Spartan 3 (b) Lattice ICE40

The construction in [39] does not use stream cipher, it uses a block cipher in counter mode to do the bulk encryption, and the suggested hash functions are either normal polynomial hashes or BRW polynomials. The performance of the construction in [39] when implemented using a AES with 128 bit key and a normal polynomial hash is shown in Table 9. The Table reports four implementations, which are described below:

| **Mode** | **Area** (Logic Cells) | **Freq.** (MHz) | **Cycles** | | **Throughput** (Mbps) | | **TPA** | | **Static power (mW)** | **Dynamic power (mW)** | **Total power (mW)** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | enc | dec | enc | dec | enc | dec | | | |
| STES[Trv,ML]-1b | 1687 | 149.31 | 13601 | 12369 | 44.92 | 49.39 | 6.50 | 7.16 | 0.16 | 32.93 | 33.89 |
| STES[Trv,ML]-4b | 1934 | 138.83 | 3401 | 3093 | 164.30 | 180.66 | 20.76 | 22.83 | 0.16 | 48.57 | 48.73 |
| STES[Trv,PD]-4b | 1991 | 140.10 | 3401 | 3093 | 168.56 | 185.35 | 20.69 | 22.75 | 0.16 | 50.87 | 50.24 |
| STES[Trv,ML]-8b | 2386 | 141.62 | 1705 | 1551 | 339.89 | 373.63 | 34.81 | 38.27 | 0.16 | 57.77 | 57.93 |
| STES[Trv,PD]-8b | 2434 | 140.44 | 1705 | 1551 | 325.92 | 358.28 | 32.72 | 35.97 | 0.16 | 59.75 | 59.91 |
| STES[Trv,ML]-16b | 3015 | 142.27 | 859 | 782 | 677.72 | 744.46 | 54.93 | 60.34 | 0.16 | 88.51 | 88.68 |
| STES[Trv,PD]-16b | 2935 | 139.26 | 859 | 782 | 648.57 | 712.44 | 54.00 | 59.32 | 0.16 | 85.74 | 85.90 |
| STES[Trv,ML]-40b | 6607 | 139.74 | 353 | 323 | 1619.87 | 1774.47 | 59.91 | 65.64 | 0.16 | 105.54 | 105.70 |
| STES[Trv,PD]-40b | 5121 | 134.78 | 353 | 323 | 1535.94 | 1682.77 | 73.30 | 80.30 | 0.16 | 170.41 | 170.57 |
| STES[Grn,ML]-1b | 2050 | 145.79 | 10337 | 9953 | 57.71 | 59.93 | 6.87 | 7.15 | 0.16 | 54.05 | 54.21 |
| STES[Grn,ML]-4b | 2147 | 138.84 | 2585 | 2489 | 219.78 | 228.26 | 25.07 | 25.98 | 0.16 | 54.38 | 54.54 |
| STES[Grn,PD]-4b | 2390 | 136.56 | 2585 | 2489 | 216.17 | 224.50 | 22.10 | 22.96 | 0.16 | 66.07 | 66.23 |
| STES[Grn,ML]-8b | 2680 | 135.75 | 1297 | 1249 | 428.29 | 444.74 | 39.05 | 40.55 | 0.16 | 59.64 | 59.80 |
| STES[Grn,PD]-8b | 2797 | 135.80 | 1297 | 1249 | 428.45 | 444.91 | 37.43 | 38.87 | 0.16 | 70.89 | 71.05 |
| STES[Grn,ML]-16b | 3788 | 136.15 | 655 | 631 | 850.57 | 882.93 | 54.87 | 56.96 | 0.16 | 104.53 | 104.96 |
| STES[Grn,PD]-16b | 3682 | 136.87 | 655 | 631 | 855.07 | 887.59 | 56.75 | 58.91 | 0.16 | 107.51 | 107.67 |
| STES[Grn,ML]-32b | 5405 | 135.29 | 332 | 320 | 1667.49 | 1730.02 | 75.39 | 78.22 | 0.16 | 171.20 | 171.36 |
| STES[Grn,PD]-32b | 5215 | 132.50 | 332 | 320 | 1633.10 | 1694.34 | 76.52 | 79.40 | 0.16 | 159.20 | 159.36 |
| STES[Mky,ML]-1b | 2794 | 143.05 | 9953 | 9665 | 58.81 | 60.56 | 5.14 | 5.30 | 0.16 | 44.65 | 44.81 |

**Table 7.** Performance of STES in Lattice ICE40. STES[SC,Hash]-$d$b denotes STES of data path $d$ instantiated with stream cipher SC and hash function Hash. Trv: Trivium, Grn: Grain128, Mky: Mickey128. ML: Multilinear universal hash, PD: Pseudo Dot Product. TPA : Throughput per area, is computed as $\frac{1}{\text{area}\times\text{time}}$, where area is in logic blocks and time in microseconds for encrypting/decrypting 512 bytes.

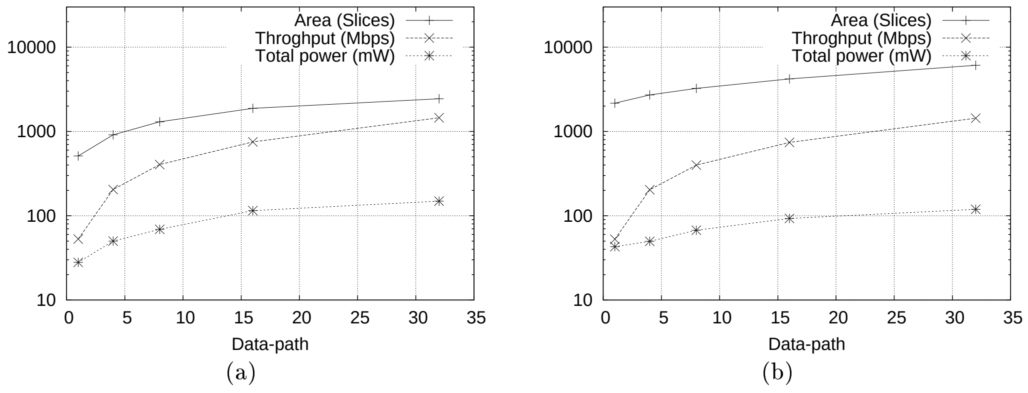

**Fig. 12.** Growth of Area, Throughput and Total power for STES[Grain,MLUH]: (a) Spartan 3 (b) Lattice ICE40

**TES-AESs-1s:** TES in [39] implemented with a sequential AES128 and a fully parallel 128 bit Karatsuba Multiplier in Spartan 3.

**TES-AESs-4s:** Sequential AES with one 4 stage pipelined 128 bit multiplier implemented in Spartan 3.

| **Mode** | **Throughput** (Mbps) | | **Static power** | **Dynamic power** | **Total power** |
|---|---|---|---|---|---|
| | enc | dec | (mW) | (mW) | (mW) |
| STES[Trv,ML]-1b | 30.08 | 33.08 | 0.16 | 22.05 | 22.21 |
| STES[Trv,ML]-4b | 120.32 | 132.30 | 0.16 | 36.27 | 36.45 |
| STES[Trv,PD]-4b | 120.32 | 132.30 | 0.16 | 34.99 | 35.15 |
| STES[Trv,ML]-8b | 240.00 | 263.83 | 0.16 | 40.79 | 40.95 |
| STES[Trv,PD]-8b | 240.00 | 263.83 | 0.16 | 42.55 | 42.70 |
| STES[Trv,ML]-16b | 476.37 | 523.27 | 0.16 | 62.22 | 62.38 |
| STES[Trv,PD]-16b | 476.37 | 523.27 | 0.16 | 61.57 | 61.73 |
| STES[Trv,ML]-40b | 1159.21 | 1270.01 | 0.16 | 75.69 | 75.85 |
| STES[Trv,PD]-40b | 1159.21 | 1270.01 | 0.16 | 126.43 | 126.59 |
| STES[Grn,ML]-1b | 39.58 | 41.11 | 0.16 | 36.30 | 36.46 |
| STES[Grn,ML]-4b | 158.29 | 164.40 | 0.16 | 38.92 | 39.08 |
| STES[Grn,PD]-4b | 158.29 | 16.44 | 0.16 | 53.89 | 54.05 |
| STES[Grn,ML]-8b | 315.50 | 327.62 | 0.16 | 42.48 | 42.64 |
| STES[Grn,PD]-8b | 315.50 | 327.62 | 0.16 | 57.41 | 57.57 |
| STES[Grn,ML]-16b | 624.73 | 648.49 | 0.16 | 79.41 | 79.57 |
| STES[Grn,PD]-16b | 624.73 | 648.50 | 0.16 | 76.64 | 76.80 |
| STES[Grn,ML]-32b | 1232.53 | 1278.75 | 0.16 | 128.12 | 128.28 |
| STES[Grn,PD]-32b | 1232.53 | 1278.75 | 0.16 | 120.15 | 120.31 |

**Table 8.** STES in Lattice ICE40 at a frequency of 100 Mhz.

**TES-AESp-4s:** 10 stage pipelined AES with 4 stage pipelined 128 bit multiplier implemented in Virtex 5.

**TES-sAES-1s:** This is an estimation based on a very compact AES reported in [34], and a polynomial hash which uses four 32 bit multipliers as used in case of MLUH-32b. The estimation is based on the data in [34] that the AES occupies 167 slices and takes 42 cycles to produce a single block of cipher. The estimated slices is obtained by summing the slices of the components and the frequency is estimated by considering that the critical path of the circuit would be given by the component with the highest critical path. Real implementations may change these data.

The results in Table 9 shows that the implementations of the TES described in [39] with a sequential AES in Spartan 3 takes up much more area than our designs with stream cipher and our designs with data path of more than 8 bits achieves higher throughput at the cost of smaller area and lower power consumption.

TES-AESp-4s is a huge design and it does not fit in a Spartan 3 device (note the slices in Virtex 5 have much more resources than the slices in Spartan 3 and the of slices in these two families are not quite comparable). TES-AESp-4s achieves throughput similar to the designs of HMCH[Poly] and HEH[Poly] reported in [15], but it cannot be in any sense considered as a light weight design. But the performance TES-AESp-4s do show that the TES in [39] can achieve quite high throughput.

The design philosophy adopted in TES-sAESs-1s is probably best comparable to our stream cipher based designs. As in TES-sAESs-1s we intend to use a very compact AES. The estimation shows that such an implementation would also occupy quite a large area but not achieve a good throughput.

| Mode | Slices | Cycles | Frequency (MHz) | Throughput (Mbps) | B-RAMS | Static power (mW) | Dynamic power (mW) | Total power (mW) |
|---|---|---|---|---|---|---|---|---|
| TES-AESs-1s Spartan 3 | 6170 | 388 | 45.58 | 480.70 | 11 | 191 | 182 | 372 |
| TES-AESs-4s Spartan 3 | 6389 | 403 | 74.07 | 752.10 | 11 | 192 | 245 | 437 |
| TES-AESp-4s Virtex 5 | 4902 | 111 | 287.44 | 10596.44 | 0 | 1243 | 2276 | 3519 |
| TES-sAESs-1s Spartan 3 * | 2800 | 2694 | 71.51 | 108.62 | 3 | - | - | - |

**Table 9.** . TES implemented using AES-128 and a polynomial hash on different platforms. TES-AESs-1s: Sequential AES with one fully parallel 128 bit multiplier, TES-AESs-4s: Sequential AES with one 4 stage pipelined 128 bit multiplier, TES-AESp-4s: Ten staged pipelined AES encryption core with 4 stage pipelined 128 bit multiplier. TES-sAESs-1s (estimation): A small AES (167 slices, 3 block RAMs, with latency of 42 cycles per block) and four 32-bit multipliers.

**Use of light weight Block Ciphers:** In the current days there have been numerous proposals for light weight block ciphers like PRESENT [10], KATAN, KTANTAN [12], KLEIN [21], LED [23] etc. These block ciphers are designed to optimize the hardware resources required to implement them. In a generic description of a block cipher based TES any secure block cipher can be used, thus there is no technical difficulty in plugging in a light weight block cipher in an existing description of a TES and this would lead to a low cost design compared to the AES alternatives that we just discussed. But a thing to note is the light weight block ciphers are mainly designed to be used in specific applications like in RFID authentication etc., and are not designed for bulk encryption in a mode. The light weight block ciphers have small block lengths, for example all the schemes mentioned above have a block length of 64 bits of lower. Such small block lengths would restrict their use in TES as the block length of the block cipher used in a block cipher based TES is an important security parameter. Recall that all existing block cipher based TES enjoys a security upper bound of $c\sigma^2/2^n$, where $\sigma$ is the query complexity of the adversary, $c$ is a small constant and $n$ the block length of the underlying block cipher. Thus, the security guarantees provided by the known reductions are not sufficient if $n$ has a value less or equal to 64. Thus we feel that given our current state of knowledge it would not be advisable to use block ciphers of small block lengths for constructing TES. It may be so that the current reductions are not tight, or there exist possibility of new constructions with better than quadratic security bounds, existing light weight block ciphers can be useful in such scenarios.

## 7.4 Discussions

The main design goal of STES was to obtain a TES which can be implemented in a compact form and would have low power consumption. Our experiments validate that STES do achieve these goals to a large extent.

In the Introduction we mentioned the speeds recommended by the SD standard. The commercially available memories does not achieve the values specified in the standard. In Table 10 we

|  | **Read Speed** (Mbps) | **Write Speed** (Mbps) | **Recommendation** Spartan 3 | ICE40 |
|---|---|---|---|---|
| Class 4 | 32 | 32 | STES[Trv,ML]-1b | STES[Trv,ML]-1b |
| Class 10 | 160 | 128 | STES[Trv,ML]-4b | STES[Trv,ML]-4b |
| Class 10 Premier | 400 | 264 | STES[Trv,PD]-8b | STES[Grn,ML]-8b |
| Class 10 Premier Pro | 760 | 360 | STES[Trv,PD]-16b | STES[Grn,PD]-16b |
| UE700 USB 3.0 | 1600 | 800 | STES[Trv,PD]-40b | STES[Grn,PD]-32b |
| S102 USB 3.0 | 800 | 400 | STES[Trv,PD]-16b | STES[Grn,PD]-16b |
| C103 USB 3.0 | 560 | 80 | STES[Trv,PD]-16b | STES[Trv,PD]-16b |

**Table 10.** Recommendations corresponding to speed rates of SD cards and USB memories of different types which are currently sold by ADATA. Our recommendations corresponds to the smallest design which provides the required performance

present the maximum speed of the various classes of memories sold by ADATA [1, 2]. The first four rows corresponds to SD cards and the last three rows gives data of USB memories. In the last two columns we mention our smallest design which can attain the necessary data rates of the given memory devices. Table 10 clearly demonstrate that STES can serve as a viable scheme for encryption in a large class of non-volatile memory devices.

## References

1. ADATA Flash Memory Cards. `http://www.adata-group.com/index.php?action=product&cid=7&lan=en&cid1=1`.
2. ADATA USB Flash Drives. `http://www.adata-group.com/index.php?action=product&cid=1&lan=en`.
3. eSTREAM, the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream/`.
4. IEEE Std 1619.2-2010: IEEE Standard for Wide-Block Encryption for Shared Storage Media. IEEE Computer Society, March 2011. Available at: `http://standards.ieee.org/findstds/standard/1619.2-2010.html`.
5. SD Association. `www.sdcard.org`.
6. Steve Babbage and Matthew Dodd. The MICKEY Stream Ciphers. In Robshaw and Billet [33], pages 191–209.
7. Mihir Bellare and Phillip Rogaway. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.
8. Côme Berbain and Henri Gilbert. On the security of IV dependent stream ciphers. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2007.
9. Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. `http://cr.yp.to/papers.html#pema`.
10. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.
11. Philippe Bulens, Kassem Kalach, FranÃğois-Xavier Standaert, and Jean-Jacques Quisquater. FPGA Implementations of eSTREAM Phase-2 Focus Candidates with Hardware Profile. 1 2007. http://sasc.cry.
12. Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.
13. Christophe De Cannière and Bart Preneel. Trivium. In Robshaw and Billet [33], pages 244–266.
14. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

15. Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodríguez-Henríquez, and Palash Sarkar. Efficient hardware implementations of BRW polynomials and tweakable enciphering schemes. *IEEE Trans. Computers*, 62(2):279–294, 2013.
16. Debrup Chakraborty and Mridul Nandi. An Improved Security Bound for HCTR. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 289–302. Springer, 2008.
17. Debrup Chakraborty and Palash Sarkar. HCH: A New Tweakable Enciphering Scheme Using the Hash-Counter-Hash Approach. *IEEE Transactions on Information Theory*, 54(4):1683–1699, 2008.
18. Sanjit Chatterjee, Alfred Menezes, and Palash Sarkar. Another look at tightness. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 293–319. Springer, 2011.
19. Martin Feldhofer. Comparison of low-power implementations of trivium and grain. In *The State of the Art of Stream Ciphers, Workshop Record*, pages 236 – 246, 2007.
20. Edgar N. Gilbert, F. Jessie MacWilliams, and Neil J. A. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53:405–424, 1974.
21. Zheng Gong, Svetla Nikova, and Yee Wei Law. KLEIN: A new family of lightweight block ciphers. In Ari Juels and Christof Paar, editors, *RFIDSec*, volume 7055 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.
22. T. Good and M. Benaissa. Hardware Results for Selected Stream Cipher Candidates. In *The State of the Art of Stream Ciphers, Workshop Record*, pages 191–204, 2007.
23. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.
24. Shai Halevi. EME$^*$: Extending EME to Handle Arbitrary-Length Messages with Associated Data. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2004.
25. Shai Halevi. Invertible Universal Hashing and the TET Encryption Mode. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 412–429. Springer, 2007.
26. Shai Halevi and Phillip Rogaway. A Tweakable Enciphering Mode. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer, 2003.
27. Shai Halevi and Phillip Rogaway. A Parallelizable Enciphering Mode. In Tatsuaki Okamoto, editor, *CT-RSA*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.
28. Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In Robshaw and Billet [33], pages 179–190.
29. David Hwang, Mark Chaney, Shashi Karanam, Nick Ton, and Kris Gaj. Comparison of FPGA-targeted hardware implementations of eSTREAM stream cipher candidates. In *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland*, pages 151–162, Feb 2008.
30. Lattice, Inc. *iCE40 Family Handbook Ultra Low-Power mobile FPGA LP, HX*, March 2012.
31. Cuauhtemoc Mancillas-López, Debrup Chakraborty, and Francisco Rodríguez-Henríquez. Reconfigurable hardware implementations of tweakable enciphering schemes. *IEEE Trans. Computers*, 59(11):1547–1561, 2010.
32. David A. McGrew and Scott R. Fluhrer. The Extended Codebook (XCB) Mode of Operation. Cryptology ePrint Archive, Report 2004/278, 2004. `http://eprint.iacr.org/`.
33. Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer, 2008.
34. Gaël Rouvroy, François-Xavier Standaert, Jean-Jacques Quisquater, and Jean-Didier Legat. Compact and efficient encryption/decryption module for fpga implementation of the aes rijndael very well suited for small embedded applications. In *ITCC (2)*, pages 583–587. IEEE Computer Society, 2004.
35. Palash Sarkar. A new multi-linear hash family. *Designs, Codes, and Cryptography.* to appear.
36. Palash Sarkar. Improving Upon the TET Mode of Operation. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 2007.
37. Palash Sarkar. Efficient Tweakable Enciphering Schemes from (Block-Wise) Universal Hash Functions. *IEEE Transactions on Information Theory.*, 55(10):4749–4760, 2009.
38. Palash Sarkar. Tweakable Enciphering Schemes From Stream Ciphers With IV. Cryptology ePrint Archive, Report 2009/321, 2009. `http://eprint.iacr.org/`.

39. Palash Sarkar. Tweakable Enciphering Schemes Using Only the Encryption Function of a Block Cipher. *Inf. Process. Lett.*, 111(19):945–955, 2011.
40. Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332, 2004.
41. Peng Wang, Dengguo Feng, and Wenling Wu. HCTR: A Variable-Input-Length Enciphering Mode. In Dengguo Feng, Dongdai Lin, and Moti Yung, editors, *CISC*, volume 3822 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2005.
42. Shmuel Winograd. A new algorithm for inner product. *IEEE Transactions on Computers*, 17:693–694, 1968.
43. Xilinx, Inc. *Spartan-3 FPGA Family Data Sheet*, December 2009.