Efficient Hardware Implementations of BRW Polynomials and Tweakable Enciphering Schemes

Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodríguez-Henríquez and Palash Sarkar*

Computer Science Department, CINVESTAV-IPN
2508 Av. IPN, San Pedro Zacatenco
Mexico City 07360, Mexico

*Applied Statistics Unit
Indian Statistical Institute,
203 B.T Road, Kolkata 700108,
India

Abstract. A new class of polynomials was introduced by Bernstein (Bernstein 2007) which were later named by Sarkar as Bernstein-Rabin-Winograd (BRW) polynomials (Sarkar 2009). For the purpose of authentication, BRW polynomials offer considerable computational advantage over usual polynomials: $(m-1)$ multiplications for usual polynomial hashing versus $\lfloor \frac{m}{2} \rfloor$ multiplications and $\lceil \log_2 m \rceil$ squarings for BRW hashing, where m is the number of message blocks to be authenticated. In this paper, we develop an efficient pipelined hardware architecture for computing BRW polynomials. The BRW polynomials have a nice recursive structure which is amenable to parallelization. While exploring efficient ways to exploit the inherent parallelism in BRW polynomials we discover some interesting combinatorial structural properties of such polynomials. These are used to design an algorithm to decide the order of the multiplications which minimizes pipeline delays. Using the nice structural properties of the BRW polynomials we present a hardware architecture for efficient computation of BRW polynomials. Finally we provide implementations of tweakable enciphering schemes proposed in Sarkar 2009 which uses BRW polynomials. This leads to the fastest known implementation of disk encryption systems.

1 Introduction

Polynomial hashes are an important part of many cryptographic protocols like message authentication codes, authenticated encryption, tweakable enciphering schemes (TES), etcetera. These schemes generally involve the computation of an univariate polynomial of degree $m-1$ defined over a finite field $\mathbb{F}_q$ as,

$$\mathsf{Poly}_h(X) = x_1 h^{m-1} + x_2 h^{m-2} + \cdots + x_{m-1} h + x_m, \tag{1}$$

where $X = (x_1, \ldots, x_m) \in \mathbb{F}_q^m$ and $h \in \mathbb{F}_q$. Traditionally, the evaluation of $\mathsf{Poly}_h(X)$ has been done using Horner's rule, which requires $(m-1)$ multiplications and $m-1$ additions in $\mathbb{F}_q$. In the rest of this paper, we will refer to $\mathsf{Poly}_h()$ as a normal polynomial.

Recently, Bernstein [1] introduced a new class of polynomials which were later named in [22] as Bernstein-Rabin-Winograd (BRW) polynomials. BRW polynomials on m message blocks defined over $\mathbb{F}_q$ have the interesting property that they can be used to provide authentication, but, unlike the normal polynomial they can be evaluated using only $\lfloor \frac{m}{2} \rfloor$ multiplications in $\mathbb{F}_q$ and $\lceil \log_2 m \rceil$ squarings. Thus, these polynomials potentially offer a computational advantage over the normal ones.

The use of BRW polynomials in hardware has not been addressed till date. As will be clear from discussions later, the structure of a BRW polynomial is fundamentally different from the normal ones, and there are some subtleties associated to their efficient implementation that are worth of further analysis.

In particular, the recursive definition of a BRW polynomial gives it a certain structure which is amenable to parallelization. It turns out that to take advantage of this parallel structure one needs to carefully schedule the order of multiplications involved in the polynomial evaluation. The scheduling is determined by the dependencies in the multiplications and also by the desired level of parallelization and hardware resources available.

The contributions of this paper are twofold. Firstly, we present a hardware architecture for efficient evaluation of BRW polynomials. The hardware design heavily depends on the careful analysis of the inherent parallelism in the structure of a BRW polynomial. This leads to a method to determine the order in which the different multiplications are to be performed.

We present an algorithm that schedules in an efficient fashion, all the $\lfloor \frac{m}{2} \rfloor$ multiplications required for the evaluation of a BRW polynomial keeping in mind the amount of parallelism desired. This algorithm leads to a hardware architecture that can perform an optimal computation of BRW polynomials in the sense that the evaluation is achieved using a minimum number of clock cycles.

As our second contribution, we present efficient hardware implementations of two TESs which use BRW polynomials. Comparisons are made with various other existing constructions which make use of normal polynomials. One of the most important applications of a TES is disk encryption. As a consequence of our implementation and comparative study, we conclude that TES schemes using BRW polynomials provide the fastest options for disk encryption.

Computing BRW polynomials in hardware: From the point of view of hardware realizations, the most crucial building block of a polynomial hash function is a field multiplier. Digit-serial multipliers yield compact designs in terms of area and enjoy short critical paths but they require several clock cycles in order to compute a single field multiplication. In contrast, fully-parallel multipliers are able to compute one field multiplication every clock cycle. However, due to their large critical path, these multipliers seriously compromise the design's maximum achievable clock frequency.

Since polynomial hash blocks require the batch computation of a relatively large number of products, it makes sense to utilize pipelined multiplier architectures. In this work, we decided to utilize a k-stage pipeline multiplier with $k = 2, 3$. After a latency period required to fill up the pipe, these architectures are able to perform one field multiplication every clock cycle. The advantage is a much shorter critical path than the one associated with fully parallel multiplier schemes [2].

In using a pipelined multiplier, our main concern is to find a proper ordering of the multiplications which would minimize the delay in the pipeline. In the ideal case there should always be multiplications ready to be done at every clock cycle. Another objective is to reduce the need to store the intermediate results so that one can minimize the extra storage locations utilized in the circuit.

To achieve this we analyze the structure of the BRW polynomial. Our analysis views the polynomial as a tree where addition and multiplication nodes are interconnected with each other. Viewing the BRW polynomial as a tree immediately gives us information about the dependence of the various operations required for its computation.

We discover some interesting properties of the tree, and use these properties to design a scheduling algorithm. The scheduling algorithm takes as input a BRW polynomial and the desired number of pipeline stages and outputs the schedule (or order) in which the different multiplications are to be performed. This schedule has several attractive features.

1. For pipeline structures with two or three stages, we give a full characterization of the number of clock cycles that is required for computing the polynomial.
2. The schedule ensures that the pipeline delays would be minimal.
3. The scheduling algorithm greedily attempts to minimize the storage. We show that the requirement of extra storage grows very slowly with the increase in the number of blocks.

Utilizing the schedule produced by the scheduling algorithm we came out with a hardware architecture that is meant for computing BRW polynomials with a fixed number of message blocks. We show-case a specific architecture which uses 31 blocks of messages and a 3-stage pipelined Karatsuba multiplier. Two variants of the architecture are discussed. In the first one, the field squaring operations are computed on the fly, whereas in the second variant the field squarings are pre-computed. Advantages and disadvantages of the two approaches are compared. Finally, we show that the design philosophy is scalable and can be utilized for different pipeline stages and different number of message blocks.

Tweakable enciphering schemes using BRW polynomials: In the second contribution of this paper, we use BRW polynomials for efficient hardware implementation of TESs. These are length preserving block-cipher modes of operations which provide security in the sense of strong pseudorandom permutations. A fully defined TES for arbitrary length messages using a block cipher was first presented in [12]. In [12] it was also first stated that a possible and important application area for such type of encryption schemes is low level disk encryption.

Since then, there has been a lot of activity towards constructions and analysis of such schemes, producing so far more than ten proposals and the hardware and software implementation of most of them.

Most TES proposals fall into three basic categories: Encrypt-Mask-Encrypt type, Hash-ECB-Hash type and Hash-Counter-Hash type. The schemes which fall within the first category use two layers of encryption with a light weight masking layer in-between. Examples are the modes CMC [12], EME [13], EME* [10].

The constructions of the other two categories use two layers of hashing with a single layer of encryption between the two hash layers. In the Hash-ECB-Hash type constructions an electronic code book mode forms the encryption layer whereas in the case of Hash-Counter-Hash constructions a counter mode of operation is used for the encryption layer. Some modes of Hash-ECB-Hash type are PEP [5], TET [11], HEH [21], whereas the modes XCB [18], HCTR [24], HCH [6], ABL [19] fall under the Hash-Counter-Hash type.

The main component of Encrypt-Mask-Encrypt type constructions are block-ciphers, and to encrypt an m block message these constructions require about $2m$ block cipher calls. On the other hand, the constructions of the type Hash-ECB-Hash and Hash-Counter-Hash require computation of two polynomial hash functions in addition to the block cipher calls. These constructions require about m block-cipher calls along with additional finite field multiplications to encrypt an m block message.

The modes that have been mentioned above use a normal polynomial evaluation, i.e., they compute the function $\mathsf{Poly}_h()$. The modes PEP, TET, HEH, HCH, HCTR, XCB all require the evaluation of two such polynomials each of them on about m blocks, thus these modes require $2m$ finite field multiplications and about m block-cipher calls. [1]

In a recent work [22], a class of new TESs was reported, which can be instantiated either by a normal polynomial or a BRW polynomial. The usage of BRW polynomial has the advantage that it can hash m blocks using about $m/2$ multiplications whereas normal polynomial evaluation would

[1] Note that the operations counts given here are approximate for the ease of discussion, see Table 1 of [17] for the exact operation counts.

require m multiplications. This decreases the computation cost significantly over the previously known modes.

Almost all known TES schemes known before [22] were implemented in various hardware platforms in [17]. In [17], a careful analysis of the possible parallelism for all the modes was done and the designs tried to exploit the schemes' parallelism to their fullest extent. The designs were targeted towards Virtex 4 family of FPGAs and the main design goal was speed. The design used a ten-stage pipelined AES encryption/decryption core. For hashing a fully parallel Karatsuba multiplier was employed for performing the field multiplications. In those modes where both block-cipher and multiplier blocks were required, the critical path was decided by the later block. The obtained throughput figures were satisfactory with the design goal which was meant to match the speed of the modern day disk controllers (the interested reader can see [17] for a detailed discussion of the design decisions and the results obtained in that work). In [17], the constructions reported in [22] were not included as these constructions are more recent.

In this work we provide efficient hardware implementations of some of the most efficient schemes reported in [22]. The fundamental difference of the schemes reported in [22] from the previous schemes is in the use of the BRW polynomials which are significantly different in structure from the normal polynomials. Using our analysis and implementation of BRW polynomials significantly brings down the length of the critical path. Further, due to the drastic reduction in the required number of field multiplications, the latency of the whole circuit also goes down. The combined effect is to provide significantly higher throughput compared to the designs studied in [17].

The constructions in [22] can also be instantiated using a normal polynomial. We compare the performance of the different instantiations. For a TES using normal polynomials we also use a pipelined multiplier and run parallel instances of the Horner's rule. Our strategy of computing a normal polynomial using pipelined multipliers is similar to the strategy used in [23].

The organization of the rest of the paper is as follows. In Section 2, we define the BRW polynomials and present a tree based analysis of such polynomials. Using the tree structure of the BRW polynomials we develop a scheduling algorithm and provide analysis of the scheduling algorithm. Finally, based on the scheduling algorithm we present the hardware architecture for computing BRW polynomials. In Section 3 we provide implementation details of the hardware architecture used for evaluating a BRW polynomial.

In Section 4, we describe the algorithms HEH and HMCH, which are the two new tweakable enciphering schemes proposed in [22]. These algorithms are analyzed from the perspective of efficient hardware implementation and specific design decisions are formulated. In Section 5, we discuss the experimental results obtained from our hardware realizations. The paper is concluded in Section 6.

2 BRW Polynomials

A special class of polynomials was introduced in [1] for fast polynomial hashing and subsequent use in message authentication codes. In [1] the origin of these polynomials were traced back to Rabin and Winograd [20], but the construction presented in [1] has subtle differences compared to the construction in [20]. The modifications were made keeping an eye to the issue of computational efficiency. Later in [22] these polynomials were used in the construction of tweakable enciphering schemes and the class of polynomials were named as Bernstein-Rabin-Winograd (BRW) polynomials.

Let $X_1, X_2, \ldots, X_m, h \in \mathbb{F}_q$, then the BRW polynomial $H_h(X_1, \ldots, X_m)$ is defined recursively as follows.

- $H_h() = 0$

- $H_h(X_1) = X_1$
- $H_h(X_1, X_2) = X_2 h + X_1$
- $H_h(X_1, X_2, X_3) = (h + X_1)(h^2 + X_2) + X_3$
- $H_h(X_1, X_2, \ldots, X_m) = H_h(X_1, \ldots, X_{t-1})(h^t + X_t) \oplus H_h(X_{t+1}, \ldots, X_m)$, if $t \in \{4, 8, 16, 32, \ldots\}$ and $t \leq m < 2t$.

Computationally the most important property is that for $m \geq 2$, $H_h(X_1, \ldots, X_m)$ can be computed using $\lfloor m/2 \rfloor$ multiplications and $\lceil \lg m \rceil$ squarings. In the rest of the paper, we will use either $H_h()$ or $\mathsf{BRW}_h()$ to denote a BRW polynomial.

2.1 A Tree Based Analysis

A BRW polynomial $H_h(X_1, \ldots, X_m)$ can be represented as a tree T_m which contains three types of nodes, namely, *multiplication nodes*, *addition nodes* and *leaf nodes*. The tree T_m will be called a BRW tree and can be recursively constructed using the following rules:

1. For $m = 2, 3$ it is easy to construct T_m directly as shown in Fig. 1.
2. If $m = 2^s$, for some $s \geq 2$, the root of T_m is a multiplication node. The left subtree of the root consists of a single addition node which in turn has the leaf nodes h^m and X_m as its left and right child, respectively. The right subtree of the root is the tree T_{m-1}.
3. If $2^s < m < 2^{s+1}$ for some $s \geq 2$, the root is an addition node with its left subtree as T_{2^s} and the right subtree as T_{m-2^s}.

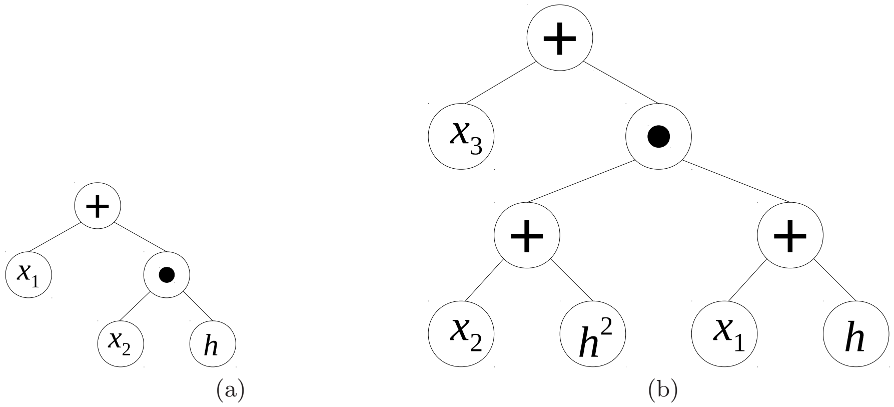


Fig. 1. Trees corresponding to $m = 2, 3$. The nodes labeled with $\odot$ and $\oplus$ represent a multiplication node and an addition node respectively. (a) Tree corresponding to $H_h(X_1, X_2)$. (b) Tree corresponding to $H_h(X_1, X_2, X_3)$.

A construction of the BRW tree T_{16} corresponding to the polynomial $H_h(X_1, \ldots, X_{16})$ is shown in Fig. 2. According to this construction, the following two properties hold.

- Any leaf node is either a message block X_j or it is h^k, for some j, k.
- For a multiplication node, either, its left child is labeled by a message block X_j and the right child is labeled by h; or, its left child is an addition node which in turn has a message block X_j and h^k as its children for some j and k. As a consequence, for a multiplication node, there is exactly one leaf node in its left subtree which is labeled by a message block.

As we are only interested in multiplications, we can ignore the addition nodes and thus simplify the BRW tree by deleting the addition nodes from it. We shall address the issue of addition later when we describe our specific design in Section 3, and we would then see that ignoring the additions as we do now will not have any significant consequences from the efficient implementation perspective. We reduce the tree T_m corresponding to the polynomial $H_h(X_1, \ldots, X_m)$ to a new tree by applying the following steps in sequence.

1. Label each multiplication node v by j where X_j is the leaf node of the left subtree rooted at v.
2. Remove all nodes and edges in the tree T_m other than the multiplication nodes.
3. If u and v are two multiplication nodes, then add an edge between u and v if u is the most recent ancestor of v in T_m.

The procedure above will delete all the addition nodes from the tree T_m. We shall call the resulting structure a *collapsed forest* (as the new structure may not be always connected, but its connected components would be trees) and denote it by F_m. Note that for every m, there is a unique BRW tree T_m and hence a unique collapsed forest F_m.

The collapsed forests corresponding to polynomials $H_h(X_1, \ldots, X_{16})$ and $H_h(X_1, \ldots, X_{30})$ are shown in Fig. 3.

By construction, the number of nodes in a collapsed forest F_m is equal to the number of multiplication nodes in T_m. The nodes of F_m are labeled with integers. Label j of a node in F_m signifies that either the multiplicands are X_j and h; or, one of the multiplicands is $(X_j + h^k)$ for some k. As a result, there is a unique multiplication associated with each node of a collapsed forest.

For example, the multiplication $(X_2 + h^2) * (X_1 + h)$ is associated to the node labeled 2 in Fig. 3. Refer to Fig. 1 to see this. Similarly, if the outputs of nodes labeled *4* and *6* are A and B respectively, then the multiplication associated with the node labeled *8* is $(X_8 + h^8) * (A + B + X_7)$.

This procedure easily generalizes and it is possible to explicitly write down the unique multiplication associated with any node of a collapsed forest. So, the problem of scheduling the multiplication in T_m reduces to obtaining an appropriate sequencing (linear ordering) of the nodes of F_m.

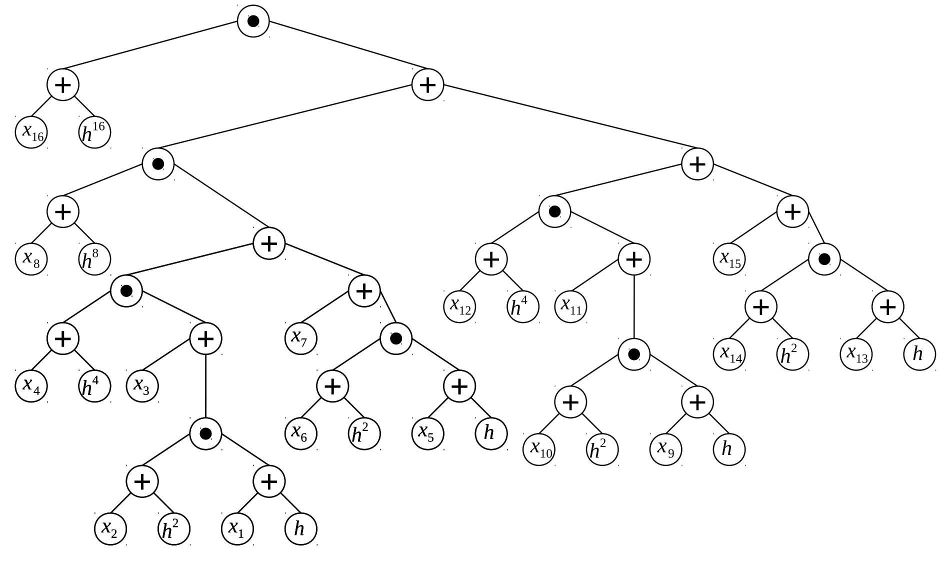
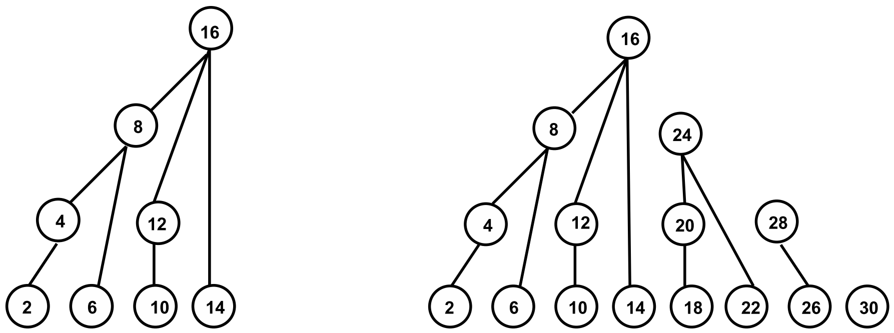


Fig. 2. The BRW tree representing $H_h(X_1, \ldots, X_{16})$.



(a) (b)

Fig. 3. (a) Collapsed forest corresponding to $H_h(X_1, \ldots X_{16})$. (b) Collapsed forest corresponding to $H_h(X_1, \ldots X_{30})$.

The structure of the collapsed forest corresponding to a polynomial $H_h(\cdot)$ helps us to visualize the dependencies of the various multiplications involved in the computation of $H_h(\cdot)$. The following definitions would help us to characterize dependencies among those operations.

Definition 1. *Let v be a node in a collapsed forest F, the level of v in F denoted by $\mathsf{level}_F(v)$ is the number of nodes present in the longest path from v to a leaf node. A node v in F such that $\mathsf{level}_F(v) = 0$ is said to be* independent. *Any node v with $\mathsf{level}_F(v) > 0$ is said to be* dependent.

Definition 2. *Suppose u, v are nodes in a collapsed forest F such that $\mathsf{level}_F(u) > \mathsf{level}_F(v)$ and u is an ancestor of v in F, then we say that u is dependent on v.*

In the following proposition, we state some important properties of collapsed forests. The proofs are a bit tedious and are obtained from the recursive structure of F_m which is in turn, inherited from the recursive structure of T_m.

Proposition 1. *Let F_m be a collapsed forest corresponding to the BRW polynomial $H_h(X_1, \ldots, X_m)$.*

1. *The number of nodes in F_m is $\lfloor \frac{m}{2} \rfloor$.*
2. *The nodes in F_m are labeled by integers $2i$, $1 \leq i \leq \lfloor \frac{m}{2} \rfloor$.*
3. *If m is even then F_m and F_{m+1} are same.*
4. *The number of connected components in F_m is equal to the hamming weight of $\lfloor \frac{m}{2} \rfloor$.*
5. *Let $p = \lfloor m/2 \rfloor$ and $\mathsf{bit}_i(p)$ denote the i^{th} bit of p where $0 \leq i \leq \mathsf{len}(p)$. If $\mathsf{bit}_i(p) = 1$ then F_m contains a tree of size 2^i.*
6. *If x is a label of a node and $x \equiv 2 \bmod 4$ then the node is an independent node.*
7. *If x is a label of a node and $x \equiv 0 \bmod 8$ then x has at least $x - 2$ and $x - 4$ as its children.*
8. *If x is the label of a node and $x \equiv 4 \bmod 8$, then $x - 2$ is the only child of x.*

2.2 Scheduling of Multiplications

Our goal, as stated earlier, is to design a circuit for computing BRW polynomials using a pipelined multiplier. If we use a pipelined multiplier with N stages, then N clock cycles would be required to complete one multiplication, but in each clock cycle N different multiplications can be processed, as long as these N multiplications happen to be independent of each other, i.e., none of these N multiplications should depend on the results of the others. Thus, if it can be guaranteed that N independent multiplications are available in each clock then the circuit will require $m + N$ clock cycles to complete m multiplications (there would be an initial latency of N clocks for filling the pipe and thereafter the result of one multiplication would be produced in each subsequent clock cycle).

A collapsed forest is a convenient way to view the dependencies among the various multiplications which are required to compute a BRW polynomial. In this section, we propose an algorithm Schedule which uses a collapsed forest to output a multiplication schedule. The aim of the algorithm is to minimize the number of clock cycles.

For designing the scheduling algorithm we require two lists L_1 and L_2. For a list L and an element x of L, we shall require the following operations.

1. Pop(L): returns the first element in L; or, returns NULL if L is empty.
2. Delete(L): deletes the first element in L.
3. Insert(x, L): inserts x in L and x becomes the last element in L.

Note that Pop(L) does not delete the first element from L. Two successive pop operations from L without any intermediate delete operation will result in the same element.

Each node in the collapsed forest is given two fields NC and ST associated with it. If x is a node in the collapsed forest then x.NC represents the number of children of node x, and x.ST denotes the time at which the node x was inserted into the list L_2 (the requirement of ST will become evident soon). Let Parent(x) denote the parent of node x in the collapsed forest.

The algorithm for scheduling is described in Fig.4. The algorithm uses a function Process which is also depicted in Fig. 4. The inputs to the algorithm are m and a variable NS which represents the number of pipeline stages. The outputs from Step 103 of Process form a sequence of integers. This provides the desired sequence of multiplications.

Before the main while loop begins (in line 11) the list L_1 contains all the independent nodes in the collapsed forest corresponding to the given polynomial and L_2 is empty. Within the while loop no nodes are inserted in L_1, but new nodes are inserted into and gets deleted from L_2. L_2 is a queue, i.e., the nodes gets deleted from L_2 in the same order as they enter it. The way we define the operations Pop(),Delete() and Insert() guarantee this.

At any given clock-cycle, the nodes in the forest can be in four possible states: *unready*, *ready*, *scheduled* and *completed*. A node x is unready if there exist a node y on which x is dependent but y has not been completed yet. A node becomes ready if all nodes on which it depends are completed. A node can only be scheduled after it is ready. Once a node is scheduled it takes NS clock cycles to get completed.

In the beginning, the nodes with level zero, i.e., the independent nodes are the only nodes in the ready state all others being in the unready state. These independent nodes are listed in L_1 at the beginning, no more nodes are further added to L_1. Thus, the nodes in L_1 can be scheduled at any time. As the algorithm proceeds, nodes gets scheduled in line 102 of the function Process.

After a node is scheduled the algorithm updates the field NC (number of children) of its parent. When the last child of a given node x is scheduled then x is inserted into the list L_2, and in the field ST of x a record of the time when its last child was scheduled is kept.

If a node is in L_2 then it is sure that all its children have been scheduled but not necessarily completed. The condition in line 12 checks if the last child of a given node in L_2 has already been completed and if a node x passes this check then it is ready to be scheduled.

For each execution of the while loop (lines 10 to 20) at most one node gets scheduled and once a node is scheduled it is deleted from the corresponding list. The condition on the while loop (line

```
**Algorithm** Schedule($m$,NS)
Construct the collapsed forest $F_m$;
**for** each node $x$ in $F_m$
    $x$.NC $\leftarrow$ number of children of $x$;
    $x$.ST $\leftarrow$ undefined;
    **if** $\text{level}_{F_m}(x) = 0$,
        Insert($x, L_1$);
**end for**
$L_2 \leftarrow$ Empty;
clock $\leftarrow$ 1;
**while** ($L_1$ and $L_2$ are both not empty)
    $x \leftarrow$ Pop($L_2$);
    **if** ($x \neq$ NULL and clock $-$ $x$.ST $>$ NS)
        Process($x, L_2$, clock);
    **else**
        $x \leftarrow$ Pop($L_1$);
        **if** ($x \neq$ NULL))
            Process($x, L_1$, clock);
    **end if**;
    clock $\leftarrow$ clock + 1;
**end while**

**Function** Process($x$, $L$,clock)
Delete($L$);
$y \leftarrow$ Parent($x$);
**Output** $x$;
**if** $y \neq$ NULL
    $y$.NC $\leftarrow$ $y$.NC $-$ 1;
    **if** ($y$.NC $= 0$)
        $y$.ST = clock;
        Insert($y, L_2$);
    **end if**;
**end if**;
**return**
```

Fig. 4. The algorithm Schedule

10) checks whether both the lists are empty and the condition on line 12 checks whether the first element of L_2 is ready, in the next two propositions we state why these checks would be sufficient.

Proposition 2. *If L_1 and L_2 are both empty then there are no nodes left to be scheduled. Further, the algorithm terminates, i.e., the condition that L_1 and L_2 are both empty is eventually attained.*

Proof. Suppose both L_1 and L_2 are empty but there is a node v which is left to be scheduled. As L_1 contains all independent nodes in the beginning and it is empty thus v is not an independent node. As v has not been scheduled and it is not in L_2 thus there must be a child of v which has not been scheduled. As there must exist a path from v to some independent node x, applying the same argument repeatedly we would conclude that there exist some independent node x which has not been scheduled. This give rise to a contradiction as L_1 is empty.

For the second statement, note that as long as L_1 is non-empty, each iteration of the while loop results in exactly one node of F_m been added to the schedule. This node is either a node in L_2 (if there is one such node), or, it is a node of L_1.

Once L_1 becomes empty, if L_2 is also empty, then by the first part, the scheduling is complete. If L_2 is non-empty, then let v be the first element of L_2. It may be possible that an iteration of the while loop does not add a node to the existing schedule. This happens if $\text{clock} - v.\mathsf{ST} \leq \mathsf{NS}$. But, the value of $v.\mathsf{ST}$ does not change while the value of clock increases. So, at some iteration, the condition $\text{clock} - v.\mathsf{ST} > \mathsf{NS}$ will be reached and the node v will be output as part of the call $\mathsf{Process}(v, L, \text{clock})$. □

Proposition 3. *If the first element of L_2 is not ready to be scheduled then no other elements in L_2 would be ready.*

Proof. Let v be the first element in L_2, as v is not ready to be scheduled, hence $clock - v.\mathsf{ST} \leq \mathsf{NS}$. Let u be any other node in L_2, as u was added to L_2 later than v thus $u.\mathsf{ST} > v.\mathsf{ST}$ and so $\text{clock} - u.\mathsf{ST} < \text{clock} - v.\mathsf{ST} < \mathsf{NS}$. Thus, u is also not ready to be scheduled. □

Example 1. We give an example of the running the algorithm for $m = 16$ and $\mathsf{NS} = 2$. The collapsed tree corresponding to the BRW polynomial $H_h(X_1, X_2, \ldots, X_{16})$ is shown in Fig. 3(a). The independent nodes in the tree are *2*, *6*, *10*, *14* and according to line 6 of the algorithm $\mathsf{Schedule}$ these nodes are inserted in the list L_1, and initially L_2 is empty. The contents of the two lists along with the output in each clock is shown in Fig. 5. The entries in the list L_2 are listed as $x(y)$, where x is the label of the node and $x.\mathsf{ST} = y$. Figure 5 shows that after clock 9 both the lists L_1 and L_2 become empty and thus the algorithm stops. There is no output produced in clock 8 as in clock 8 L_1 is empty and the only node in L_2 is not ready as its start time is 7, which means that its ultimate child got scheduled in clock 7 and thus is yet to be completed. The following sequence of nodes is produced as output of $\mathsf{Schedule}$.

2, *6*, *4*, *10*, *8*, *12*, *14*, *16*.

We describe the scheduling of multiplications corresponding to this sequence. Again refer to Fig. 2.

M_1: $R_1 = (X_2 + h^2)(X_1 + h)$;
M_2: $R_2 = (X_6 + h^2)(X_5 + h)$;
M_3: $R_3 = (X_4 + h^4)(X_3 + R_1)$;
M_4: $R_4 = (X_{10} + h^2)(X_9 + h)$;
M_5: $R_5 = (X_8 + h^8)(R_3 + R_2 + X_7)$;
M_6: $R_6 = (X_{12} + h^4)(X_{11} + R_4)$;
M_7: $R_7 = (X_{14} + h^2)(X_{13} + h)$;
M_8: $R_8 = (X_{16} + h^{16})(R_5 + R_6 + R_7 + X_{15})$.

The 8 multiplications are $M_1, \ldots, M_8$. In this example, we have not tried to minimize the number of intermediate storage registers that are required. A method for doing this will be discussed later. Note the following points.

1. In each of the multiplications, the subscript of X in the first multiplicand is the label of the corresponding node in F_{16}.
2. The scheduling is compatible with $\mathsf{NS} = 2$, i.e., a 2-stage pipeline: M_3 and M_4 depend on the output of M_1 and so start 2 clocks after M_1 starts; M_5 depends on the output of M_2 and M_3 and starts 2 clocks after M_3; and so on.

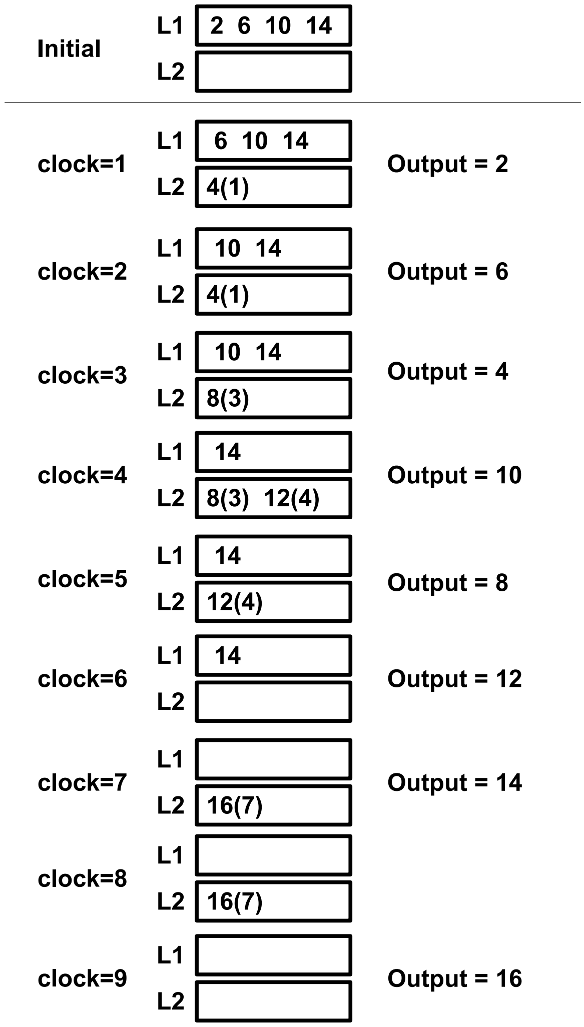


Fig. 5. The states of the lists L_1 and L_2 when $\mathsf{Schedule}(16, 2)$ is run. The entries in the list are denoted as $x(y)$ where x is the label of a node and $y = x.\mathsf{ST}$

The output of the algorithm Schedule for various number of blocks for $\mathsf{NS} = 2$ and 3 are shown in Table 1. The entries $-$ in Table 1 means that no multiplication was scheduled in the corresponding clock. The last column (total clocks) is the clock when the last multiplication was scheduled.

Table 1. The output of Schedule for NS $= 2, 3$ for small number of blocks

Number of pipeline stages NS=2

Blocks	Clock															Total
(m)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	clocks
2	*2*															1
4	*2*	–	*4*													3
6	*2*	*6*	*4*													3
8	*2*	*6*	*4*	–	*8*											5
10	*2*	*6*	*4*	*10*	*8*											5
12	*2*	*6*	*4*	*10*	*8*	*12*										6
14	*2*	*6*	*4*	*10*	*8*	*12*	*14*									7
16	*2*	*6*	*4*	*10*	*8*	*12*	*14*	–	*16*							9
18	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*							9
20	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*						10
22	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*					11
24	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	–	*24*			13
26	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*			13
28	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*	*28*		14
30	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*	*28*	*30*	15

Number of pipeline stages NS=3

Blocks	Clock															Total
(m)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	clocks
2	*2*															1
4	*2*	–	–	*4*												4
6	*2*	*6*	–	*4*												4
8	*2*	*6*	–	*4*	–	–	*8*									7
10	*2*	*6*	*10*	*4*	–	–	*8*									7
12	*2*	*6*	*10*	*4*	–	*12*	*8*									7
14	*2*	*6*	*10*	*4*	*14*	*12*	*8*									7
16	*2*	*6*	*10*	*4*	*14*	*12*	*8*	–	–	*16*						10
18	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	–	*16*						10
20	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	–	*16*	*20*					11
22	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*					11
24	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	–	–	*24*		14
26	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	*26*	–	*24*		14
28	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	*26*	–	*24*	*28*	15
30	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	*26*	*30*	*24*	*28*	15

2.3 Optimal Scheduling

Given a BRW polynomial on m message blocks, the number of nodes in the corresponding collapsed tree is $p = \lfloor m/2 \rfloor$. The scheduling of these nodes is said to be *optimal* if one node can be scheduled in each clock-cycle thus requiring p clock-cycles to schedule all the nodes. If such a scheduling is possible for a given value of the number of stages (NS) we say that the scheduling admits a *full pipeline*, as such a scheduling will not give rise to any pipeline delays.

An optimal scheduling will not exist for all values of m and NS. Existence of an optimal scheduling for NS stages means that in each clock cycle NS independent nodes are available.

If m is a power of two then it is easy to see that the collapsed forest would contain a single tree and the root would be dependent on all other nodes (as is the case in Fig. 3(a)), thus no scheduling procedure can yield an optimal scheduling for such an m for any NS > 1.

Also, as the number of pipeline stages increases, for an optimal scheduling to be possible, more independent multiplications are required. For small values of NS, however, the following theorem gives the conditions for which Schedule gives an optimal scheduling for NS $= 2$ and 3.

Theorem 1. *Let $H_h(X_1, X_2, \ldots, X_m)$ be a BRW polynomial and let $p = \lfloor m/2 \rfloor$ be the number of nodes in the corresponding collapsed forest. Let clks be the number of clock cycles taken by* Schedule *to schedule all nodes, then,*

1. *If* NS $= 2$*, and $p \geq 3$, then*
$$clks = \begin{cases} p+1 & \text{if } p \equiv 0 \bmod 4; \\ p & \text{otherwise.} \end{cases}$$
2. *If* NS $= 3$ *and $p \geq 7$, then*
$$clks = \begin{cases} p+2 & \text{if } p \equiv 0 \bmod 4; \\ p+1 & \text{if } p \equiv 1 \bmod 4; \\ p+1 & \text{if } p \equiv 2 \bmod 4; \\ p & \text{if } p \equiv 3 \bmod 4. \end{cases}$$

Proof. Both the proofs are by induction. We present the proof only for $\mathsf{NS} = 2$ as the other case is similar. For $p = 3$ (i.e. $m = 6$) the explicit output of the algorithm is shown in Table 1, which proves that the base case is true. Suppose the results hold for some $p \geq 3$ and we wish to show the results for $p + 1$. There are the following cases to consider:

1. $p + 1 \equiv 1 \bmod 4$. Then $p \equiv 0 \bmod 4$, hence by induction hypothesis the p nodes were scheduled in $p + 1$ cycles, signifying that there was one cycle when no node was scheduled. The last node in this case has label $2(p + 1)$ and as $2(p + 1) \equiv 2 \bmod 4$, hence the last node is an independent node (from Proposition 1), hence the last node can be scheduled in the missed cycle, thus the total clocks required for $p + 1$ nodes would be $p + 1$.
2. $p + 1 \equiv 2 \bmod 4$. Then, $p \equiv 1 \bmod 4$, hence by induction hypothesis p nodes were scheduled in p cycles, the last node to be scheduled has label $2(p + 1)$ and $2(p + 1) \equiv 4 \bmod 8$ and hence by Proposition 1, has only one child and the label of the child is $2p$. Considering the previous case, $2p$ was not the last node to be scheduled; hence, the node $2(p + 1)$ can be scheduled in the $p + 1$-th cycle.
3. $p + 1 \equiv 3 \bmod 4$. Then, $p \equiv 1 \bmod 4$, hence p nodes were scheduled in p cycles, the last node to be scheduled has label $2(p + 1)$ and $2(p + 1) \equiv 2 \bmod 4$ and hence by following the same arguments as in case 1 the nodes can be scheduled in $p + 1$ cycles.
4. $p + 1 \equiv 0 \bmod 4$. Then, $p \equiv 3 \bmod 4$, hence by induction hypothesis p nodes were scheduled in p cycles. Considering cases 2 and 3 if p nodes are scheduled then the last node to be scheduled has label $2(p - 2)$ which is a child of the node $2(p + 1)$, hence the node $2(p + 1)$ cannot be scheduled in the $p + 1$-th cycle. Thus the number of cycles required would be $p + 2$.

This completes the proof. □

From the proof above one can obtain a recursive description of the output of the scheduling algorithm for $\mathsf{NS} = 2$. Let $p \geq 4$, and $x_1, \ldots, x_p$ be the sequence for p, where $x_1, \ldots, x_p \in \{2, 4, \ldots, 2p\}$. Then, the following is the construction of the sequence for $p + 1$:

If $p + 1 \equiv 0 \bmod 2$ then output the sequence $x_1, \ldots, x_p, 2(p + 1)$;
If $p + 1 \equiv 3 \bmod 4$, then output the sequence $x_1, \ldots, x_p, 2(p + 1)$;
If $p + 1 \equiv 1 \bmod 4$, then output the sequence $x_1, \ldots, x_{p-1}, 2(p + 1), x_p$.

Similarly if $\mathsf{NS} = 3$, and if $x_1, \ldots, x_p$ be the sequence for $p \geq 6$, then the following is the construction of the sequence for $p + 1$:

If $p + 1 \equiv 0 \bmod 2$, then output the sequence $x_1, \ldots, x_p, 2(p + 1)$;
if $p + 1 \equiv 1 \bmod 4$, then output the sequence $x_1, \ldots, x_{p-2}, x_{p-1}, 2(p + 1), x_p$;
if $p + 1 \equiv 3 \bmod 4$, then output the sequence $x_1, \ldots, x_{p-2}, 2(p + 1), x_{p-1}, x_p$.

2.4 The Issue of Extra Storage

Optimizing the number of clock cycles should not be the only goal for a scheduling algorithm. An important resource associated with a pipelined architecture is the requirement of extra storages for storing the intermediate results. The issue of storage in the case of computing BRW polynomials is simple, we illustrate the issue with an example. Refer to the diagram of the collapsed tree in Fig. 3(b), suppose for a two-stage pipeline we schedule the multiplications in the following order:

$$\mathit{2, 6, 10, 14, 18, 22, 26, 30, 4, 12, 20, 28, 8, 24, 16} \tag{2}$$

This schedule requires 15 clock cycles and is thus optimal, but this is very different from the order of the multiplications given by the algorithm Schedule. This ordering, though it is optimal in the terms of number of clock cycles required, requires more extra storage for storing the intermediate results. Recall that the dependence of the nodes in the BRW tree shows that multiplication operation represented by a node x may be started when all its children have been completed. In each clock cycle at most one multiplication gets completed, thus the intermediate results computed for the children of x have to be stored, as they will be required for the computing of x. If the scheduling is done as in Eq. (2) then the starting times and finishing times (in clocks) of the nodes would be as below.

Nodes	*2*	*6*	*10*	*14*	*18*	*22*	*26*	*30*	*4*	*12*	*20*	*28*	*8*	*24*	*16*
Starting Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Finishing Time	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Note that the results of the multiplications in nodes *2*, *10*, *18*, *26* which are completed in the clocks 3, 5, 7 and 9, are further used to compute the multiplications in the nodes *4*, *12*, *20* and *28* which are started in the clocks 9, 10, 11 and 12 respectively. Hence, the results obtained in the clocks 3, 5, 7 and 9 are all needed to be stored. If we continue in this manner we shall see that the scheduling in Eq. (2) would require a significant amount of extra storage for storing the intermediate results.

In contrast to the scheduling in Eq. 2, if we follow the algorithm Schedule, then the starting and the finishing time of the nodes would be as:

Nodes	*2*	$\underline{\mathit{6}}$	*4*	*10*	$\underline{\mathit{8}}$	$\underline{\mathit{12}}$	*14*	*18*	*16*	$\underline{\mathit{20}}$	*22*	*26*	*24*	*28*	*30*
Starting Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Finishing Time	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Number of intermediate storages for this schedule is just one and can be seen from the following considerations.

- Node *2* is completed in clock 3 and in the same clock node *4* gets started which requires the result of the multiplication in clock 3 thus the result of node *2* is not required to be stored.
- In clock 4 node *6* is completed and *10* is started, as *10* does not depend on *6*, hence the result of node *6* needs to be stored.
- Continuing in this way we see that only the results of nodes *6*, *8*, *12* and *20* are needed to be stored (they are underlined in the table above).
- But, this does not mean that four distinct storage locations are required, as the storage locations can be reused.
- Note that node *8* is ready in clock 7 and it is required to be stored. Node *6* was stored previously, and the result was already utilized when node *8* started in clock 5. Thus the location used for storing *6* can be used to store *8*.
- Arguing in this manner the total number of storage locations required in this case is just 1.

Determining the number of intermediate storage locations required by Schedule. The design of the algorithm Schedule tries to minimize the requirement of extra storage by trying to use the intermediate results as quickly as possible. For any given input, the extra storage requirements of Schedule can be easily determined from the following two simple principles.

1. A result x is required to be stored if it is completed in a certain clock t and the node y which starts at t is not a parent of x.
2. If there exists a storage location which stores results that have been already used, then the location can be reused, otherwise a new storage location must be defined.

The extra storage requirement for Schedule grows very slowly with the increase in the number of message blocks. Figure 6 shows the number of storage for various number of message blocks for $\mathsf{NS} = 3$.

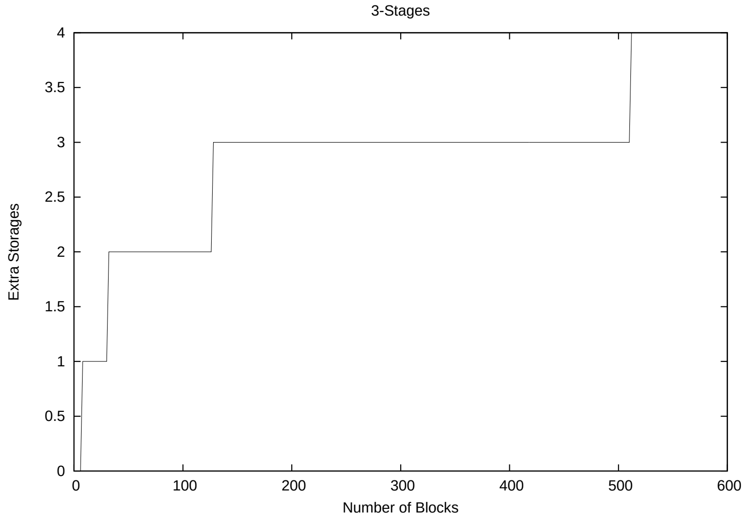


Fig. 6. The growth of number of extra storage locations required with the number of blocks for $\mathsf{NS} = 3$.

3 A Hardware Architecture for the Efficient Evaluation of BRW Polynomials

Utilizing the nice properties of the BRW polynomials as discussed in the previous sections we propose a hardware architecture for computing such polynomials. We "show-case" our architecture for 31 blocks of messages using a three-stage pipelined multiplier. The number of message blocks of the polynomial and the pipeline stages of the multiplier can be varied without hampering the design philosophy. This issue of scalability is discussed later.

Each block is 128 bits long, and so the multiplication, addition and squaring operations take place in the field $\mathbb{F}_{2^{128}}$ generated by the irreducible polynomial $\tau(x) = x^{128} + x^7 + x^2 + x + 1$. This specific design would be also useful for the designing of tweakable enciphering schemes which are discussed in Section 4.

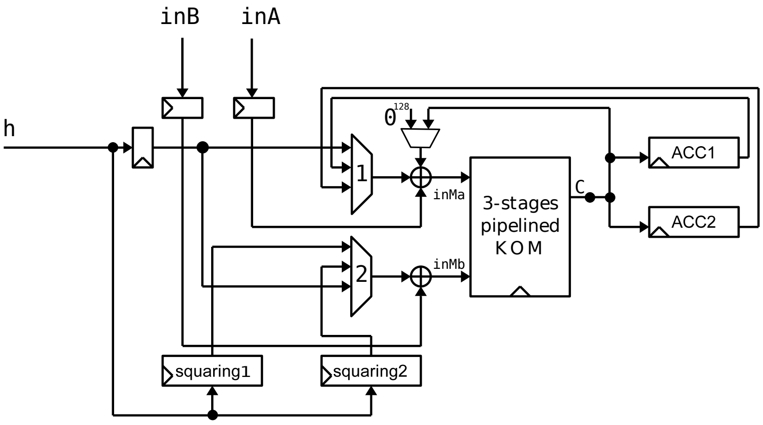


Fig. 7. Architecture for computing the BRW polynomial.

The schematic diagram of the proposed architecture is shown in Fig. 7, where the principal component is a three-stage pipelined Karatsuba multiplier denoted as KOM. (We postpone the detailed design of the multiplier to Appendix A). At the output of the multiplier, we placed two accumulators, ACC1 and ACC2, which are used to accumulate intermediate results.

Figure 7 also includes two blocks for computing squares in the field $\mathbb{F}_{2^{128}}$. These circuits are depicted in the diagram as squaring1 and squaring2. Computing squares in binary extension fields are much easier than multiplications. The strategy used for computing squares is as follows.

Let $\alpha \in \mathbb{F}_{2^{128}}$. Then, α can be seen as a polynomial $\alpha = \sum_{i=0}^{127} b_i x^i$, where each $b_i \in \{0, 1\}$. Then

$$\alpha^2 = \left(\sum_{i=0}^{127} b_i x^i \right)^2 \bmod \tau(x) = \sum_{i=0}^{127} b_i x^{2i} \bmod \tau(x).$$

Both squaring blocks in Fig. 7 are equipped with output registers that allow to save the last field squaring computation. The multiplier block KOM has two inputs designated as inMa and inMb.

The first multiplier input (inMa) is the field addition of three values. Explanations of these values are as follows.

1. The first of these values is the output of a multiplexer block MUX1 that selects between the key h or any one of the two accumulators.
2. The second value is the output of another multiplexer that selects between the last output produced by the multiplier or zero.
3. Finally, the third value is the input signal inA.

The second multiplier input (inMb) consists of the field addition of two values. Explanations of these values are as follows.

1. The first one is taken from the output of a multiplexer MUX2 that selects either the output of squaring1, or squaring2 or the key h.
2. The second value is the input inB.

As was discussed in Section 2, the computation of a 31-block BRW polynomial denoted as, $H_h(P_1, \ldots, P_{31})$, requires the calculation of $\lfloor \frac{31}{2} \rfloor = 15$ multiplications. We give in Fig. 8 the time diagram that specifies the way that these fifteen multiplications were scheduled. The final value of the polynomial $H_h(P_1, \ldots, P_{31})$ is obtained in just eighteen clock cycles.

The dataflow specifics of the architecture in Fig. 7 is shown in the time diagram of Fig. 8. This figure shows the different data stored/produced in the various blocks at each clock cycle along with the order in which the multiplications were performed. $M_1, \ldots, M_{15}$ denote the fifteen multiplications to be computed and the multiplicands are depicted in the rows designated inMa and inMb, which are the two inputs of the KOM block.

The row designated C denotes the output of the multiplier. As a three-stage pipelined multiplier is being used, a multiplication scheduled at clock i can be obtained at C in clock $i + 3$.

The rows ACC1 and ACC2 denote the values which are accumulated in the accumulators in the various clock cycles. Note that an entry M_i in any of the rows representing the state of the two accumulators signify that the value M_i gets xor-ed to the current value in the accumulator, and an entry $*M_i$ denotes that the accumulator gets initialized by M_i.

The rows squaring1 and squaring2 show the state of the squaring circuits output register. Each of the circuits for squaring can compute the square of the current content of the output register in one clock cycle, maintain its current state, or initialize its value with h^2 taking h as a fresh input.

As depicted in Fig. 8, the computation of the polynomial $H_h(X_1, \ldots, X_{31})$ can be completed in 18 clock cycles and the final value can be obtained from the accumulator ACC2.

The circuit shown in Fig. 7 uses the strategy of computing the squares as required on the fly. An alternative strategy would be to pre-compute the required powers of h and store them in registers. By using this strategy we can get rid of the squaring circuits at the cost of some extra storage, and come up with a circuit which would be very similar to the circuit described in Fig. 7.

If the pre-computing strategy is adopted, then for computing $H_h(P_1, \ldots, P_{31})$ we need to store h^2, h^4, h^8, h^{16} in registers. The multiplexer which feeds inMb in this case would be a five-input multiplexed, where four of the inputs come from the registers where the squares were stored and the fifth input is the input line h. As squaring in binary extension fields is easy, these two strategies do not provide significantly different performances. This becomes evident from the experimental results.

Irrespective of the way in which squarings are performed, the construction of the circuit follows the scheduling strategy as dictated by the algorithm Schedule. According to Theorem 1, if a three-stage pipelined multiplier is used, then for computing $H_h(P_1, \ldots, P_{31})$ the 15 multiplications can be scheduled in 15 clock cycles without any pipeline bubbles.

Figure 8 shows that this is indeed the case as starting from clock 1 to 15, in each clock, a multiplication gets scheduled without any pipeline delays. The extra storage required to store the intermediate products is provided by the accumulator ACC1, which stores the products M_2, M_5, M_6 and M_9.

ACC2 is used to accumulate the final result, note that the products M_{10}, M_{13}, M_{14} and M_{15} are accumulated in order in the accumulator ACC2. These multiplications corresponds to the nodes *16*, *30*, *24*, *28* of the collapsed forest (see Fig. 3(b)), which in turn are the roots of the trees.

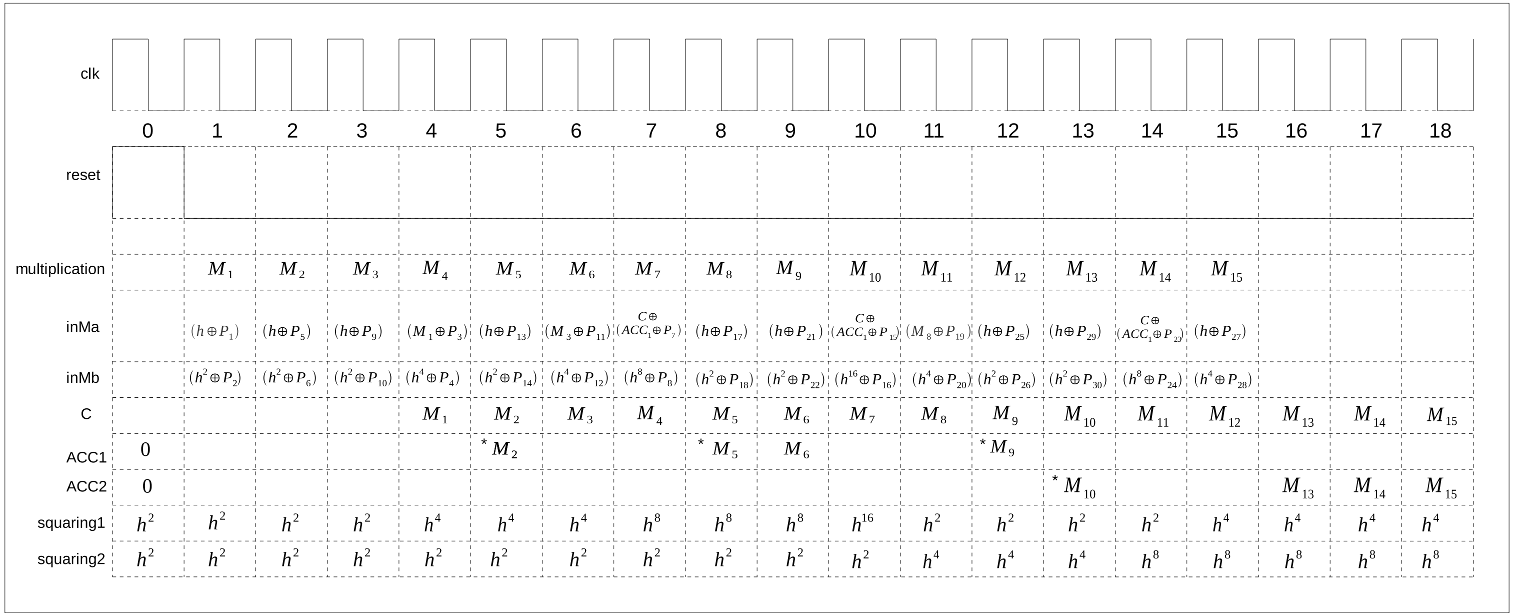
Scalability. The architecture presented previously is meant for 31-block messages. But the same design philosophy can be used for k-block messages for any fixed k.

Here we give a short description of how the circuit for computing $H_h(P_1, \ldots, P_m)$ grows with the growth of m. A 3-stage pipelined multiplier is assumed. For ease of exposition, we shall only consider the case where the powers of h are pre-computed.

The main components of the circuit will be the two multiplexers which are connected to the inputs of the multiplier, the accumulators and the registers to store the powers of h. If $H_h(P_1, \ldots, P_m)$ is to be computed, then we will require to store $h^2, h^4, \ldots, h^{2^s}$ where $2^s \leq m < 2^{s+1}$. This will require s registers.

$MUX2$ would thus be a $(s+1)$-input multiplexer. The number of accumulators required would be at most one more than the number of extra storages required. For a given polynomial $H_h(P_1, \ldots, P_m)$, the number of extra storages required by Schedule can be determined using the procedure described in Section 2.4.

If the number of accumulators required is α then MUX1 would be substituted by an $(\alpha+1)$-input multiplexer, where α inputs come from the accumulators and the last one is the input line h. The dataflow specifics can be automatically obtained from the algorithm Schedule.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
multiplication		M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}			
inMa		$(h\oplus P_1)$	$(h\oplus P_5)$	$(h\oplus P_9)$	$(M_1\oplus P_3)$	$(h\oplus P_{13})$	$(M_3\oplus P_{11})$	$C\oplus$ $(ACC_1\oplus P_7)$	$(h\oplus P_{17})$	$(h\oplus P_{21})$	$C\oplus$ $(ACC_1\oplus P_{15})$	$(M_8\oplus P_{19})$	$(h\oplus P_{25})$	$(h\oplus P_{29})$	$C\oplus$ $(ACC_1\oplus P_{23})$	$(h\oplus P_{27})$			
inMb		$(h^2\oplus P_2)$	$(h^2\oplus P_6)$	$(h^2\oplus P_{10})$	$(h^4\oplus P_4)$	$(h^2\oplus P_{14})$	$(h^4\oplus P_{12})$	$(h^8\oplus P_8)$	$(h^2\oplus P_{18})$	$(h^2\oplus P_{22})$	$(h^{16}\oplus P_{16})$	$(h^4\oplus P_{20})$	$(h^2\oplus P_{26})$	$(h^2\oplus P_{30})$	$(h^8\oplus P_{24})$	$(h^4\oplus P_{28})$			
C					M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}
ACC1	0					$^*M_2$			$^*M_5$	M_6			$^*M_9$						
ACC2	0													$^*M_{10}$			M_{13}	M_{14}	M_{15}
squaring1	h^2	h^2	h^2	h^2	h^4	h^4	h^4	h^8	h^8	h^8	h^{16}	h^2	h^2	h^2	h^2	h^4	h^4	h^4	h^4
squaring2	h^2	h^2	h^2	h^2	h^2	h^2	h^2	h^2	h^2	h^2	h^2	h^4	h^4	h^4	h^8	h^8	h^8	h^8	h^8

Fig. 8. Time diagram of the circuit in Fig. 7.

4 TES constructions based on BRW Polynomials

We shall devote this section to study an application of BRW polynomial for construction of a cryptographically useful object. As stated in the Introduction, in a recent work [22] it was suggested that BRW polynomials can be used instead of normal polynomials to design tweakable enciphering schemes of the hash-ECB-hash and hash-counter-hash family. Tweakable enciphering schemes are known to be useful in design of in-place disk encryption scheme, and in the light of the present standardizing activities of IEEE working group on security in storage the study of these schemes has gained much importance in the current days. In [22] it was claimed that TES constructions using BRW polynomials would be far more efficient than their counter parts which use normal polynomials. The claim was justified using operation counts, as a BRW polynomial requires about half the amount of multiplications than the normal polynomials. But, in [22] real design issues were not considered and thus there exist no hard experimental data to demonstrate the amount of speedups which can be achieved by the use of such polynomials. Here we concentrate on the real design issues for hardware implementation of some of the schemes described in [22], and ultimately provide experimental results which justifies that TES with BRW polynomials would have higher throughput than the ones using the normal ones.

4.1 The Schemes

There are two basic schemes described in [22], which are named as HEH and HMCH. The schemes can be instantiated in different ways for different applications. The encryption and decryption algorithms for HEH and HMCH are described in Figures 9 and 10 respectively. The descriptions are for a specific instantiation which is suitable for the purpose of disk encryption.

In the description of the algorithms we assume that $E_K : \{0,1\}^n \rightarrow \{0,1\}^n$ is a block cipher, whose inverse is $E_K^{-1} : \{0,1\}^n \rightarrow \{0,1\}^n$. The additions and multiplications are all in the field $\mathbb{F}_{2^n}$ represented by a irreducible polynomial $\tau(x)$ of degree n which is primitive. For our implementations we use the field $\mathbb{F}_{2^{128}}$ and $\tau(x) = x^{128} + x^7 + x^2 + x + 1$. An $A \in \{0,1\}^n$ can be seen as a polynomial $a_0 + a_1x + \cdots \oplus a_nx^{n-1}$ where each $a_i \in \{0,1\}$, thus every n bit string A can be treated as an element in $\mathbb{F}_{2^n}$. By xA we mean the n bit binary string corresponding to the polynomial $x(a_0 + a_1x + \cdots + a_nx^{n-1}) \bmod \tau(x)$. This operation can be performed easily by a shift and a conditional xor. In the description $\psi_h(.)$ can be instantiated in two different ways, it can either be $h\mathsf{Poly}_h(.)$ or $hH_h(.)$, where $H_h(.)$ is a BRW polynomial. From now onwards to avoid confusion we shall represent a BRW polynomial by $\mathsf{BRW}_h(.)$, and for the two different instantiations we shall call the schemes as HEH[BRW], HEH[Poly] and HMCH[BRW], HMCH[Poly].

4.2 Analysis of the Schemes and Design Decisions

We analyze here the schemes presented in Section 4 from the perspective of efficient hardware implementations and thus come up with some basic strategies for designing them. The implementation is targeted towards the disk encryption application, thus in the following discussions we shall only consider messages of fixed lengths which are 512 byte long, i.e. 32 blocks of 128 bits. [2] Our primary design goal is speed, but we shall try to keep the area metric reasonable. The basic components of both schemes are a block cipher (which we chose to instantiate using AES-128) and the polynomial hash (either Poly or BRW). Thus, in terms of hardware the basic components required would be an AES (both encryption and decryption cores) and an efficient finite-field multiplier. As the focus of

[2] 512 byte is the current size of disc sectors, though there are proposals that in the coming days we would have sector sizes 4096 bytes long, so the basic strategy of design that we shall present would have the required scalability.

Fig. 9. Encryption and decryption using HEH.

```
**Algorithm** HEH.Encrypt$^T_{h,K}(P_1, \ldots, P_m)$

$\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;
$U \leftarrow P_m \oplus \psi_h(P_1, \ldots, P_{m-1})$;
$PP_m \leftarrow U \oplus \beta_1$;
$CC_m \leftarrow E_K(PP_m)$; $V \leftarrow CC_m \oplus \beta_2$;
**for** $i \leftarrow 1$ to $m - 1$,
    $PP_i = P_i \oplus U \oplus x^i\beta_1$;
    $CC_i \leftarrow E_K(PP_i)$;
    $C_i \leftarrow CC_i \oplus x^i\beta_2 \oplus V$;
**end for**
$C_m \leftarrow V \oplus \psi_h(C_1, \ldots, C_{m-1})$;
**return** $(C_1, \ldots, C_m)$;
```

```
**Algorithm** HEH.Decrypt$^T_{h,K}(C_1, \ldots, C_m)$

$\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;
$U \leftarrow C_m \oplus \psi_h(C_1, \ldots, C_{m-1})$;
$CC_m \leftarrow U \oplus \beta_2$;
$PP_m \leftarrow E_K^{-1}(CC_m)$; $V \leftarrow PP_m \oplus \beta_1$;
**for** $i \leftarrow 1$ to $m - 1$,
    $CC_i = C_i \oplus U \oplus x^i\beta_2$;
    $PP_i \leftarrow E_K^{-1}(CC_i)$;
    $P_i \leftarrow PP_i \oplus x^i\beta_1 \oplus V$;
**end for**
$P_m \leftarrow V \oplus \psi(P_1, \ldots, P_{m-1})$;
**return** $(P_1, \ldots, P_m)$;
```

Fig. 10. Encryption and decryption using HMCH.

```
**Algorithm** HMCH.Encrypt$^T_{h,K}(P_1, \ldots, P_m)$

$\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;
$M_1 \leftarrow P_1 \oplus \psi_h(P_2, \ldots, P_m)$;
$U_1 \leftarrow E_K(M_1)$ ; $S \leftarrow M_1 \oplus U_1 \oplus \beta_1 \oplus \beta_2$;
**for** $i = 2$ to $m$,
    $C_i \leftarrow P_i \oplus E_K(x^{i-2}\beta_1 \oplus S)$ ;
**end for**
$C_1 \leftarrow U_1 \oplus \psi_h(C_2, \ldots, C_m)$;
**return** $(C_1, \ldots, C_m)$;
```

```
**Algorithm** HMCH.Decrypt$^T_{h,K}(C_1, \ldots, C_m)$

$\beta_1 = E_K(T)$; $\beta_2 = x\beta_1$;
$U_1 \leftarrow C_1 \oplus \psi_h(C_2, \ldots, C_m)$;
$M_1 \leftarrow E_K^{-1}(U_1)$ ; $S \leftarrow M_1 \oplus U_1 \oplus \beta_1 \oplus \beta_2$;
**for** $i = 2$ to $m$,
    $P_i \leftarrow C_i \oplus E_K(x^{i-2}\beta_1 \oplus S)$ ;
**end for**
$P_1 \leftarrow M_1 \oplus \psi_h(P_2, \ldots, P_m)$;
**return** $(P_1, P_2, \ldots, P_m)$;
```

this work is in BRW polynomials, in the rest of this Section we shall discuss about the instantiation with only BRW polynomials here, the instantiation with $\mathsf{Poly}_h()$ is briefly discussed in Section 4.5.

Referring to the algorithm $\text{HEH.Encrypt}_{h,K}^{T}$ of Fig. 9, we see that irrespective of the choice of $\psi_h(.)$, $(m+1)$ encryption calls to the block-cipher are required, whereas $\text{HEH.Decrypt}_{h,K}^{T}$ requires one encryption call and m decryption calls to the block cipher. The encryption/decryption calls in lines 4 and 7 of both HEH.Encrypt and HEH.Decrypt procedures are independent of each other and thus can be suitably parallelized. Algorithm $\text{HMCH.Encrypt}_{h,K}^{T}$ of Fig. 10, requires $(m+1)$ encryption calls to the block-cipher, and for $\text{HMCH.Decrypt}_{h,K}^{T}$, m encryption calls and one decryption call to the block-cipher are required. The $(m-1)$ block-cipher calls required by both encryption and decryption procedures of HMCH can be parallelized. Thus, for both modes the bulk amount of block-cipher calls can be parallelized. This suggests that a pipelined implementation of AES would be useful for implementing the ECB mode in HEH and the counter type mode in HMCH. Computation of the $\mathsf{BRW}_h(.)$ can also be suitably parallelized (as discussed in Section 3). Thus we also decided to use a pipelined multiplier to compute the BRW hash.

Out of many possible AES designs reported in the literature [16, 9, 4, 14, 7] we decided to implement a 10-stage pipelined AES core architecture with the counter mode and/or the electronic code book functionalities. This decision was taken based on the fact that the structure of the AES algorithm admits to a natural ten-stage pipeline design, where after 11 clock cycles one can get an encrypted block in each subsequent clock-cycle. We refrain ourselves from using deeper pipeline designs such as the ones reported in [15], because such designs would incur a higher latency, i.e., the total delay before a single block of cipher-text can be produced would be higher with more pipeline stages. As the message lengths in the target application are particularly small (512 bytes), such pipeline designs are not suitable for a disk sector encryption application.

As a target device for the implementation we choose FPGAs of the Virtex 5 family. These are one of the most efficient devices available in market. In [3] a highly optimized AES design suitable for Virtex 5 FPGAs was reported. One important design decision taken in [3] was to implement the byte substitution table using the LUT fabric, this is in contrast to previous AES designs where extra block RAMs were used for the storage of the look up tables. This change has a positive impact both in area and the length of the critical path, given rise to better performance. The design described in [3] is sequential. The AES design implemented in this work closely follows the techniques used in [3], but we suitably adapt and extend the techniques in [3] to a pipelined design. Moreover, another important characteristic of our AES design is that we do not attempt to design a single core for the encryption and decryption functionalities but instead, we chose to design separate cores for encryption and decryption. This gives us better throughput and also provides some extra flexibility in terms of optimization.

As it has been mentioned, in the case of the field multiplier we decided to use a three stage pipelined Karatsuba multiplier. The number of stages was fixed keeping an eye to the critical path of the circuit. Once we fixed our design for AES we selected the pipeline stages for the multiplier in such a manner that it matches the critical path of the AES. As both components would be used in the circuit, hence if a very high number of pipeline stages for the multiplier is selected then, the critical path would be given by the AES but the latency for multiplication would increase. Several exploratory experiments suggested that a three stage pipeline would be optimal as the critical path of such a circuit would just match that of the AES circuit.

Both HEH$[\psi]$ and HMCH$[\psi]$ were proved to be secure as tweakable enciphering schemes in [22]. The security proof requires $\psi_h()$ to be a almost xor universal (AXU) hash function. Both $h\mathsf{BRW}(X_1, \ldots, X_{m-1})$ and $h\mathsf{Poly}(X_1, \ldots, X_{m-1})$ are AXU. If $\pi : \{1, \ldots, m-1\} \rightarrow \{1, \ldots, m-1\}$ be a fixed permutation then it is easy to see that $h\mathsf{BRW}(X_{\pi(1)}, X_{\pi(2)}, \ldots, X_{\pi(m-1)})$ would also be AXU. Thus, using any fixed ordering of the messages for evaluating each of the BRW polynomials in the modes will not hamper their security properties. This observation is important in the context of hardware implementations of HEH[BRW] and HMCH[BRW]. As, for optimal computation of BRW polynomials we require a different order of the messages than the normal order. In our case, the permutation $\pi()$ is dictated by the algorithm $\mathsf{Schedule}$. If $m = 31$ and the number of pipeline stages of the multiplier is 3 the permutation π as dictated by $\mathsf{Schedule}$ is shown below.

The permutation $\pi(x)$

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi(x)$	1	2	7	8	3	4	13	14	5	6	11	12	9	10	19	20	15	16	21	22	17	18	27	28	23	24	29	30	25	26	31

Thus, for implementing HEH[BRW].Encrypt we replace $\psi_h(P_1, \ldots, P_{31})$ in line 2 of Fig. 9 by $h\mathsf{BRW}(P_{\pi(1)}, \ldots, P_{\pi(31)})$. Similar change is done in line 10 of the encryption algorithm and lines 2 and 10 of the decryption algorithm. For implementing HMCH[BRW] we replace $\psi_h(P_2, \ldots, P_{32})$ in line 2 of Fig. 10 by $h\mathsf{BRW}(P_{\pi(1)+1}, P_{\pi(2)+1}, \ldots, P_{\pi(31)+1})$. Similar change is done in line 10 of the encryption algorithm and lines 2 and 10 of the decryption algorithm.

4.3 Analysis of the schemes

With these basic design decisions as described above, we shall analyze HEH and HMCH to exploit the maximum parallelization possible. The following discussion assumes the use of $h\mathsf{BRW}(.)$ in place of $\psi_h(.)$ and the number of blocks to be 32 for both the schemes. First we analyze HEH which is described in Fig. 9. In Line 2 of the encryption algorithm the computation of the BRW polynomial on 31 blocks takes place. Using a 3 stage pipelined multiplier and the design described in Section 3, $\mathsf{BRW}(P_1, \ldots, P_{31})$ can be completed in 18 clock cycles and computation of $h\mathsf{BRW}(P_1, \ldots, P_{31})$ would thus require 21 clock cycles for the extra multiplication with h. Thus the computation of U (as in line 2) can be completed in 21 clock cycles. The computation of β_1 and β_2 (in line 1) can be done in parallel with the computation of U.

Then in lines 4 to 9 the main operations required are 32 calls to AES. Following our design these 32 calls can be completed in 43 cycles, and after an initial delay of 11 cycles we shall obtain one value of C_i $(i < m)$ in each cycle. For computing C_m we again need to compute the BRW polynomial which would take 21 cycles. The computation of the BRW polynomial can be parallelized with the block cipher calls, as soon as we start getting outputs of the AES calls we can start computing the BRW polynomial necessary in line 10. The specific architecture that we have designed for the BRW polynomials requires the availability of two input blocks per each clock cycle. For this reason we decided to have two AES cores running in parallel which can feed the circuit for computing the BRW polynomials and thus can reduce the total latency of the circuit. Using this strategy, all values of CC_i would be produced in 27 cycles instead of 43. After 11 of these 27 cycles we can start computing the BRW polynomial and would require a total of 21 cycles to complete. The total computation can be completed in 55 cycles if two AES cores are used. Decryption would be similar, but we need to implement two AES decryption cores for obtaining the same latency as encryption.

If we use a single AES core, then we would not be able to do the BRW computation in line 11 in 21 cycles as in each cycle we shall not be able to obtain two blocks of data as required, thus for each multiplication we need to wait two cycles, and thus the total computation for the second hash (in line 11) would require 35 cycles, and the total computation would require 69 cycles.

In case of HMCH also line 2 can be completed in 21 cycles and line 1 can be performed in parallel with line 2. For computing line 3 which involves a single AES call we would need to wait 11 cycles. Again, using two ten staged pipelined AES encryption cores the computation in line 5, which involves 31 calls to the AES in counter mode can be completed in 27 cycles. After 11 of these

27 cycles, the BRW hash can be started and it would require 21 cycles to compute. Thus, the total computation could be done in 66 cycles. In case of decryption there is only one inverse call to the AES as in line 3. Thus, for decryption there is no need to implement two AES decryption cores as is required in case of HEH. Only one decryption core is sufficient in this case and also as there is only one call, a pipelined design for this core is also un-necessary. Hence, we designed a sequential decryption core which saved us some area. If a single AES core is used, as in the case of HEH in HMCH also an additional 14 cycles would be required for computing the second hash.

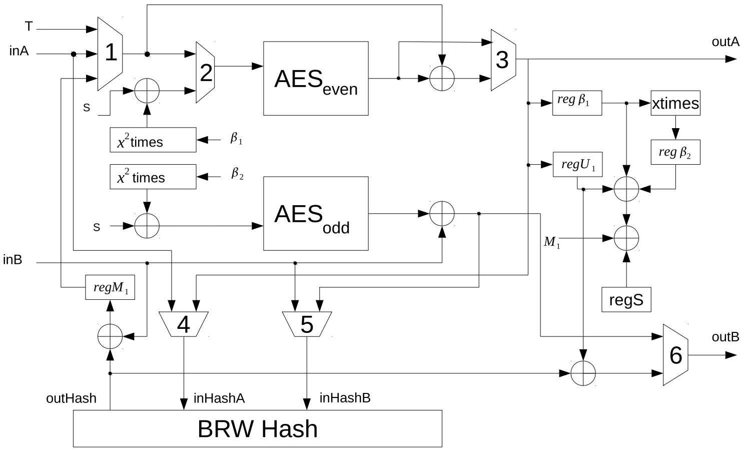


Fig. 11. Architecture for performing the HMCH[BRW] Encryption Scheme in hardware

4.4 Architecture of HMCH[BRW]

We implemented the modes HEH[BRW] and HMCH[BRW]. For both the modes both encryption and decryption functionality were implemented in a single chip. In this section we shall only describe the architecture for HMCH[BRW] which uses two pipelined encryption cores and a single sequential decryption core. The simplified architecture for HMCH[BRW] is depicted in Fig. 11. For ease of exposition in Fig. 11 we only show the encryption part of the circuit, an additional component of the circuit is the sequential decryption core which we omit for the sake of simplicity. The main components of the general architecture depicted in Fig. 11 are the following: A BRW polynomial hash block (which corresponds to the circuit shown in Fig. 7), two AES cores (equipped with both electronic code book and counter mode functionalities), and two $x^2\,Times$ blocks. The $x^2\,Times$ blocks compute x^2A, where $A \in \mathbb{F}_{2^{128}}$. The architecture also includes five registers to store the values M_1, β_1, β_2, U_1 and S, and makes use of six multiplexer blocks labeled 1 to 6 in the figure and we shall refer to them as MUX1 to MUX6. When the $x^2\,Times$ block is first activated, it simply outputs the value placed at its input (for the circuit of Fig. 11, this input value will correspond to either β_1 or β_2). Thereafter, at each clock cycle the field element x^2A will be produced as an output, where $A \in \mathbb{F}_q$ is the last value computed by this block. The control unit of this architecture consists of a ROM memory where a microprogram with sixty seven micro-instructions has been stored, each microinstruction consisting of 28-bit control words. Additionally, the control unit uses a counter that gives the address of the next instruction to be executed.

The general dataflow of Fig. 11 can be described as follows. First the parameter β_1 is computed as $\beta_1 = E_K(T)$. This is done by properly selecting MUX1 and MUX2 so that the tweak T gets encrypted in single mode by the AES$_{even}$ core. The value so obtained is stored in the register $reg\beta_1$ and also $\beta_2 = x\beta_2$ is computed and stored in $reg\beta_2$. Then, the plaintext blocks $P_2, \ldots, P_m$ are fed into the BRW hash block through the inputs *inA* and *inB* and the proper selection of MUX4 and MUX5. After 21 clock cycles, the hash of the plaintext blocks is available at *outHash*, allowing the computation of the parameter M_1 as, $M_1 = outHash \oplus P_1$, where P_1 is taken from the input signal *inB*. The parameter U_1 is computed as $E_K(M_1)$ by selecting the third input of MUX1 as the input value for the AES$_{even}$ core. The value so computed is stored in *reg*U_1. At this point the circuit of Fig. 11 is ready to compute the encryption in counter mode of $m-1$ plaintext blocks (corresponding to line 5 of the HMCH[BRW] encryption algorithm shown in Fig. 10) as,

$$\begin{aligned} AES_{even} : C_i &\leftarrow P_i \oplus E_K(x^{i-2}\beta \oplus S), \text{ for } i = 2, 4, \ldots, 31. \\ AES_{odd} : C_i &\leftarrow P_i \oplus E_K(x^{i-2}\beta \oplus S), \text{ for } i = 3, 5, \ldots, 32. \end{aligned}$$

It is noticed that this last computation is achieved in 28 clock cycles using the two AES cores in parallel. The encryption blocks C_i for $i = 2, \ldots, m$ are simultaneously sent to circuit's outputs *outA* and *outB*, and to the BRW hash block through a proper selection of MUX4 and MUX5. After 21 clock cycles, the cipher blocks' hash is available at *outHash*, allowing the computation of the encryption block C_1 as, $C_1 = outHash \oplus U_1$, where U_1 was previously computed and stored as explained above.

4.5 HEH[Poly] and HMCH[Poly]

For the sake of comparison we also implemented HEH[Poly] and HMCH[Poly]. As stated in Section 4 these schemes can be obtained by replacing $\psi_h()$ by $\mathsf{Poly}_h()$ in the algorithms of Figures 9 and 10. When a normal polynomial is used for the constructions then the usual Horner's rule is the most efficient way to compute it. At first glance, the advantages of a pipelined multiplier cannot be used due to the sequential nature of the Horner's rule. In [23] A three way parallelization strategy was proposed to evaluate a normal polynomial using three different multipliers and thus running three different instances of the Horner's rule in parallel. We adopt the strategy presented in [23] by utilizing a three staged pipelined multiplier as a tool to evaluate a normal polynomial using Horner's rule.

As we are interested in encrypting 32 blocks of messages hence in case of both HEH[Poly] and HMCH[Poly] the polynomial to be computed is

$$\begin{aligned} \psi_h(P_1, \ldots, P_{31}) &= h\mathsf{Poly}_h(P_1, P_2, \ldots, P_{31}) \\ &= h\sum_{i=1}^{31} P_i h^{31-i} \\ &= h(p_1 + p_2 + p_3) \end{aligned}$$

where

$$\begin{aligned} p_1 &= \sum_{i=1}^{11} P_{3i-2}(h^3)^{11-i} \\ p_2 &= h^2\sum_{i=1}^{10} P_{3i-1}(h^3)^{10-i} \\ p_3 &= h\sum_{i=1}^{10} P_{3i}(h^3)^{10-i}. \end{aligned}$$

Note that the multiplications in p_1 does not depend on the multiplications in p_2 and p_3, etc. Hence, a three staged pipelined multiplier can be used to compute $h\mathsf{Poly}_h(P_1, P_2, \ldots, P_{31})$. If h^2 and h^3 are pre-computed then the computation of the polynomial can be completed in 35 clock cycles.

For HEH[BRW] we used two pipelined AES encryption and decryption cores and for HMCH[BRW] we used two pipelined encryption core and a single sequential decryption core. The usage of two AES cores gave us considerable savings in the number of clock cycles as discussed in Section 4.3, as $h.\mathsf{BRW}(.)$ could be computed in only 21 cycles. But $h.\mathsf{Poly}(.)$ requires 35 clock cycles to complete, and hence dedicating two cores for this task does not give rise to any savings. Hence, while implementing HEH[Poly], we used one pipelined AES encryption core and one pipelined AES decryption core and for HMCH[Poly] we used one pipelined AES encryption core and one sequential AES decryption core.

5 Experimental Results

In this section we present the experimental results obtained form our implementations. All reported results were obtained from place and route simulations, where the target device is XILINX Virtex 5 xc5vlx330-2ff1760. Table 2 shows the performance of the basic primitives. Table 2 clearly shows that $\mathsf{BRW}_h(.)$ is far better in performance than $\mathsf{Poly}_h(.)$, but $\mathsf{BRW}_h(.)$ occupies more slices than $\mathsf{Poly}_h(.)$. We note that only the pipelined AES decryption core achieved lower frequency than the hash blocks. Thus in case of HMCH[BRW], which does not use the pipelined decryption core, the critical path is given by the hash block and in case of HEH[BRW] the critical path is given by the pipelined decryption core.

Table 2. Primitive operations on Virtex-5 device.

Core	Slices	Cycles	Frequency MHz	Throughput Gbits/Sec
AES pipelined encryption (AES-PEC)	2859	1	300.56	38.471
AES pipelined decryption (AES-PDC)	3110	1	239.34	30.720
AES sequential decryption (AES-SDC)	1075	11	292.483	3.403
$h\mathsf{Poly}_h(P_1, \ldots, P_{31})$	1886	35	251.383	28.499
$h\mathsf{BRW}_h(P_1, \ldots, P_{31})$	2086	21	243.487	46.007

For both HEH[BRW] and HMCH[BRW] we implemented three variants, we name these variants as 1, 2 and 3. The naming conventions along with the performance of the variants are described in Table 3. Table 3 also shows the variants using Poly. From the results shown in Table 3 we can infer the following:

1. **BRW versus Poly:** The variants using BRW give better throughput but occupies more area than the variants using Poly.
2. **Single core versus double core:** When two AES cores are used for HEH[BRW] and HMCH[BRW] the throughput is much higher than the case when one AES core is used, as using two AES cores we can accommodate more parallelization. In particular, the following observations can be made:
   - For the two core implementations we gain 14 clock cycles against the one core implementations. The improvement in clock cycles (66 versus 80 in case of HMCH[BRW] ; or 55 versus 69 in case of HEH[BRW]) is not reflected to that extent in the throughput (13 versus 11 in case of HMCH[BRW]; or 15 versus 13 in case of HEH[BRW])). This is due to operation at lower frequencies for the double-core implementations.
   - Increase in hardware for HMCH[BRW]-1 over HMCH[BRW]-3 is probably not significant, but, for HEH[BRW] the increase is marked. The reason behind this is for HEH[BRW]-1 two pipelined AES decryption cores are also necessary for achieving the desired parallelization.

3. **Pre-computing squares versus computing squares on the fly:** Pre-computing squares for BRW polynomials gives a negligible improvement on throughput and the circuits using pre-computation utilizes a few slices more than the circuits where squares are computed on the fly.
4. **HEH versus HMCH:**
   - HEH[BRW] gives better throughput than HMCH[BRW]. The reason being the increased latency in case of HMCH[BRW]. HMCH[BRW] has an AES call (the one in line 3 of Fig. 10) which cannot be parallelized. This results in an additional latency of 11 cycles in HMCH[BRW] compared to HEH[BRW].
   - For the same reason HEH[Poly] gives better throughput than HMCH[Poly].
   - HEH[BRW] requires pipelined AES decryption cores for the required parallelization in decryption but for HMCH[BRW] decryption a sequential AES decryption core is sufficient. Thus, HMCH[BRW] occupy lesser area than HEH[BRW].
   - HEH[BRW]-3 is comparable to HMCH[BRW]-1 and HMCH[BRW]-2 both in terms of number of slices and throughput.
5. **Recommendation:**
   - For best speed performance, use double-core HEH[BRW]; in particular, HEH[BRW]-2.
   - For smallest area, use HMCH[Poly].
   - For best area-time measure, use single-core HMCH[BRW], i.e., HMCH[BRW]-3.

Table 3. Modes of operation on Virtex-5 device. AES-PEC: AES pipelined encryption core, AES-PDC: AES pipelined decryption core, AES-SDC: AES sequential decryption core, SOF : squares computed on the fly, SPC: squares pre-computed

Mode	**Implementation Details**	**Slices**	**Frequency (MHz)**	**Clock Cycles**	**Time (nS)**	**Throughput (Gbits/Sec)**	$\frac{1}{(\text{Slice}*\text{Time})}$
HMCH[BRW]-1	2 AES-PEC, 1 AES-SDC, SOF	8040	211.785	66	311.637	13.143	399.112
HMCH[BRW]-2	2 AES-PEC, 1 AES-SDC, SPC	8140	212.589	66	310.458	13.193	395.706
HMCH[BRW]-3	1 AES-PEC, 1 AES-SDC, SOF	6112	223.364	80	358.160	11.436	**456.814**
HEH[BRW]-1	2 AES-PEC, 2 AES-PDC, SOF	11850	202.856	55	271.128	15.170	311.248
HEH[BRW]-2	2 AES-PEC, 2 AES-PDC, SPC	12002	203.894	55	269.748	**15.184**	308.879
HEH[BRW]-3	1 AES-PEC, 1 AES-PDC, SOF	8012	218.384	69	315.957	12.964	395.020
HMCH[Poly]	1 AES-PEC, 1 AES-SDC	**5345**	225.485	94	416.879	9.825	448.789
HEH[Poly]	1 AES-PEC, 1 AES-PDC	6962	218.198	83	380.388	10.768	377.606

6 Conclusion

We studied BRW polynomials from a hardware implementation perspective and designed an efficient architecture to evaluate BRW polynomials. The design of the architecture was based on a combinatorial analysis of the structural properties of BRW polynomials. Our experiments show that BRW polynomial to be an efficient alternative for normal polynomials. Moreover we explored constructions of hardware architectures for tweakable enciphering schemes using BRW polynomials and the results show that using BRW polynomials are a far better alternative to normal polynomials in terms of speed for design of TES.

References

1. Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. http://cr.yp.to/papers.html#pema.
2. Jean-Luc Beuchat, Jérémie Detrey, Nicolas Estibals, Eiji Okamoto, and Francisco Rodríguez-Henríquez. Fast architectures for the η_t pairing over small-characteristic supersingular elliptic curves. *Computers, IEEE Transactions on*, 60(2):266 –281, feb. 2011.
3. Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gaël Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In Serge Vaudenay, editor, *AFRICACRYPT*, volume 5023 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2008.
4. D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.
5. Debrup Chakraborty and Palash Sarkar. A New Mode of Encryption Providing a Tweakable Strong Pseudo-random Permutation. In Matthew J. B. Robshaw, editor, *Fast Software Encryption 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2006.
6. 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.
7. Y. Fu, L. Hao, and X. Zhang. Design of an Extremely High Performance Counter Mode AES Reconfigurable Processor. In *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05)*, pages 262–268. IEEE Computer Society, 2005.
8. Kris Gaj and Pawel Chodowiec. FPGA and ASIC implementations of AES. In Cetin Kaya Koc, editor, *Cryptographic Engineering*, pages 235–294. Springer, 2009.
9. T. Good and M. Benaissa. AES on FPGA from the Fastest to the Smallest. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2005.
10. 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.
11. 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.
12. Shai Halevi and Phillip Rogaway. A Tweakable Enciphering Mode. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer, 2003.
13. Shai Halevi and Phillip Rogaway. A Parallelizable Enciphering Mode. In Tatsuaki Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.
14. S. F. Hsiao and M. C. Chen. Efficient Substructure Sharing Methods for Optimising the Inner-Product Operations in Rijndael Advanced Encryption Standard. *IEE Proceedings on Computer and Digital Technology*, 152(5):653–665, September 2005.
15. K. Jarvinen, M. Tommiska, and J. Skytta. Comparative Survey of High-Performance Cryptographic Algorithm Implementations on FPGAs. *Information Security, IEE Proceedings*, 152(1):3–12, October 2005.
16. E. López-Trejo, F. Rodríguez Henríquez, and A. Díaz-Pérez. An Efficient FPGA Implementation of CCM Mode Using AES. In *International Conference on Information Security and Cryptology*, volume 3935 of *Lecture Notes in Computer Science*, pages 208–215, Seoul, Korea, December 2005. Springer-Verlag.
17. Cuauhtemoc Mancillas-Lopez, Debrup Chakraborty, and Francisco Rodriguez-Henriquez. Reconfigurable hardware implementations of tweakable enciphering schemes. *IEEE Transactions on Computers*, 59:1547–1561, 2010.
18. 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/.
19. David A. McGrew and John Viega. Arbitrary Block Length Mode, 2004. Available at: http://grouper.ieee.org/groups/1619/email/pdf00005.pdf.
20. Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433458, 1972.

21. 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.
22. Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions.. *IEEE Transactions on Information Theory.*, 55(10):4749–4760, 2009.
23. Akashi Satoh, Takeshi Sugawara, and Takafumi Aoki. High-performance hardware architectures for galois counter mode. *IEEE Transactions on Computers*, 54(7):917–930, 2009.
24. 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.

Appendix A: Implementation Details

The Multiplier

A Karatsuba multiplier computes the polynomial product $c = A \cdot B$, for $A = A^L + A^H x^{64}$, and $B = B^H + B^H x^{64} \in \mathbb{F}_{2^{128}}$ as,

$$C = A^L B^L + \left[(A^H + A^L)(B^L + B^H) - (A^H B^H + A^L B^L))\right] x^{64} + A^H B^H x^{128}.$$

With a computational cost of three 64-bit polynomial multiplications and 4 additions/subtractions. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to half of its previous value. After 7 iterations of applying this recursive strategy, all the polynomial operands collapse into single coefficients. However, it is common practice to stop the Karatsuba recursion earlier, performing multiplications with small operands using alternative techniques that are more compact and/or faster.

Both TES schemes discussed in this work, namely, MCTR[BRW] and HEH[BRW], require the computation of two BRW polynomial hash functions. In this work we implemented that block using a three-stage pipelined Karatsuba multiplier, whose architecture is shown in Fig. 12. Due to its recursive formulation, Karatsuba multipliers proved to be especially suitable for pipelined architecture constructions.

The multiplier shown in Fig. 12 uses three 64-bit Karatsuba multipliers, and in turn, each one of them are composed by three 32-bit multipliers and successively we implemented 16-bit and 8-bit multiplier blocks. We decided to stop the Karatsuba recursion at 4-bit level, where we used a schoolbook multiplier. After a careful timing analysis we decided to place registers at the output of the three 64-bit multipliers, at the output of the 8-bit multiplier and finally, after the 128-bit reduction block. This gives us a three-stage pipelined multiplier with each one of its three stages balanced in terms of their critical path. In fact, the critical path of the first stage is shorter than the other two stages because we wanted to include the critical path associated to the input multiplexer block (see Fig. 7) as a part of the critical path associated with the other two stages of our Karatsuba multiplier architecture.

The AES

We designed the AES encryption and decryption cores separately. For HEH[BRW] we used two pipelined AES encryption cores and two pipelined AES decryption cores and for HMCH[BRW] we used two pipelined AES encryption core and one sequential decryption core. The AES design closely follows the techniques used in [3]. In [3] the S-boxes were implemented as 256 × 8 multiplexers. This was possible due to special six input lookup tables (LUT) available in Virtex 5 devices. One S-box fits

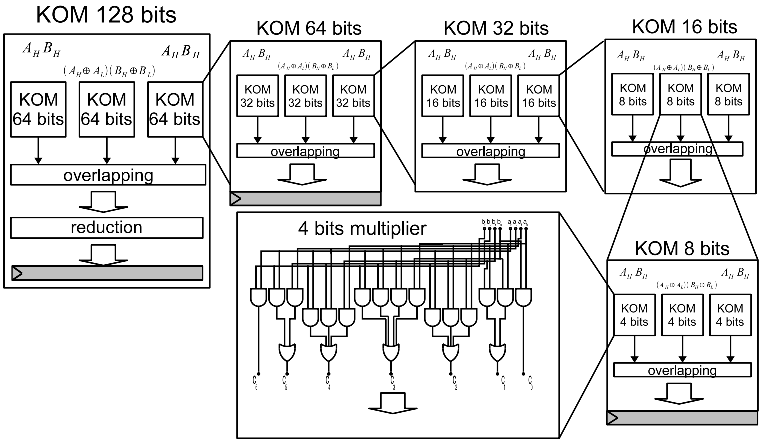


Fig. 12. Architecture of the three-stage pipelined Karatsuba multiplier

in 32 six inputs LUTs available in Virtex 5 FPGA devices. In [3] the authors presented a sequential core, we extended their idea to a $10 - stages$ pipelined core. Initially, both the encryption and decryption cores take 10 clock cycles to produce a valid output, and produces one block as output in subsequent cycles. In Fig. 13we show the basic strategy. AES encryption/decryption consist of 10 rounds, the rounds 1 to 9 has four transformations: SubBytes (BS), ShiftRows (SR), MixColumns (MC) and AddRoundKey (ADDRK). The last round has only three transformations BS, SR and ADDRK. The decryption core looks similar to figure 13, where each transformation is replaced by its inverse and the order of rounds are inverted (i.e., the computation starts at round 10 and the final output is given by the xor of initial key and output of round 1).

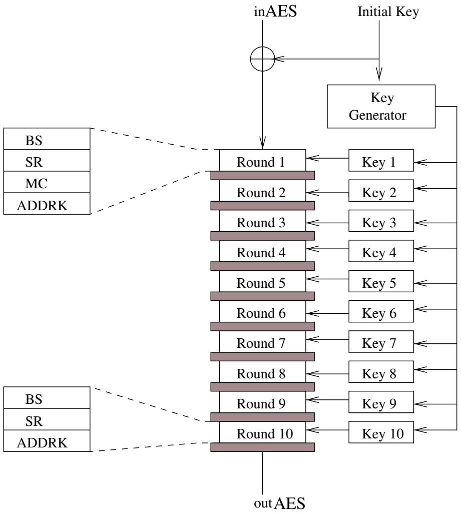


Fig. 13. Architecture for 10-stages pipelined encryption AES core.

The sequential decryption core in HMCH[BRW] was implemented using the techniques in [8]. In [8] the operations BS and MC are combined in a substitution operation which the authors call as T-boxes. In our implementation for the decryption core we used inverse T-boxes (iTbox) which combines the operations inverse byte substitution (iBS) and inverse mixcolumn (iMC). We implemented the T-boxes using large multiplexers and avoided the use of memories. As a sequential core was only required hence we needed to implement only two rounds (see Fig. 14). Irrespective of the number of cores used we used a single key generator and the S-boxes for the key generator were also implemented using multiplexers.

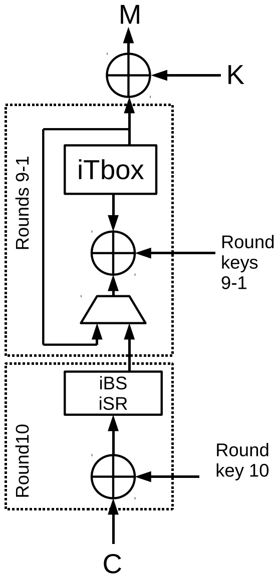


Fig. 14. Architecture for sequential decryption AES core.