High-speed architectures for binary-tree based stream ciphers:

Leviathan case study

Abstract
Real-time applications such as streaming media and voice require encryption algorithms that do not propagate errors and support fast encryption on small blocks. Since IP packets are delivered out-of-order in routed networks it is difficult to synchronize the source and the destination, therefore requiring encryption algorithms to support out-of-order generation of key stream. In this paper we investigate high-speed parallel and pipeline architectures for Leviathan, a binary tree based synchronous stream cipher that does not propagate errors and supports ordered and out-of-order key stream generation. The parallel architecture bounds the worst-case variance in the time between consecutive key words and the pipeline architecture generates key words at a uniform rate. In order to eliminate the storage overhead associated with straightforward preorder traversal, we developed a novel traversal scheme that intertwines inorder traversal with preorder traversal. We implemented pipeline architecture using Xilinx FPGAs to support 1 Gbps key stream generation rate.
1 Introduction

The Internet is a connectionless routed network built of interconnected nodes known as routers and end-hosts. The Internet Protocol (IP) is used by nodes in the internet to communicate with each other. Routers forward traffic to end-hosts and to other routers based on IP packet header information. Although the IP packet header information is sufficient to route packets, it does not provide for the privacy of the packet data. In order to guarantee the privacy of the packet data the Internet has an overlay of Virtual Private Networks (VPN) [1]. There are two commonly used methods for building VPNs—trusted VPN and secure VPN. In a trusted VPN the provider guarantees that no other traffic has access to the links providing service. In a secure VPN the nodes communicate by encrypting the information. Secure VPNs are primarily built using the IP Security Protocol (IPSec) [2].

Deployment of IPSec based VPN has accelerated the demand for high-speed cryptographic architectures supporting keyed hash functions, public-key algorithms, private-key block ciphers and stream ciphers [3]. Hash functions translate arbitrary length messages into a fixed length digest. Asymmetric public-key algorithms use one-way functions and separate encryption and decryption keys; one cannot derive the decryption key from the publicly available encryption key. On the other hand, symmetric private-key algorithms use a single key for encryption and decryption. While private-key block ciphers operate on large blocks of plaintext (ciphertext), stream ciphers operate on blocks as small as a single bit. Examples of block ciphers include Data Encryption Standard (DES) [4] and Advanced Encryption Standard (AES) [5].

![Figure 1: Key stream generator: The output function, the next state function and the state differentiate stream cipher implementations. The key words in the key stream, the plaintext and the ciphertext have the same length.](image)

Stream ciphers generate a pseudorandom key stream from a smaller seed key and are practical approximations to the one-time pad [6]. A one-time pad is a random key used only once to encrypt a plaintext of equal length. The pseudorandom key stream is xored with the plaintext stream to generate a ciphertext stream [7]. Figure 1 shows key stream generation by a stream cipher. While a synchronous stream cipher does not consider the ciphertext while generating the key stream, an asynchronous stream cipher does. Consequently, unlike an asynchronous stream cipher, a synchronous stream does not propagate errors. In a synchronous stream cipher both the encrypting and the decrypting entities should be in the same state. This requires that the key stream be generated in strict ascending numerical byte order. For the IP based routed networks in which the packets arrive out-of-order this amounts to generation of key stream for packets that have yet to arrive in order to process the packets that have already arrived. Linear Feedback Shift Register (LFSR) based synchronous stream ciphers [8] and block ciphers operating in a stream cipher mode [9],[10][11] do not have the ability to generate the key stream out-of-order.

Leviathan is a binary tree based synchronous stream cipher that has the ability to seek to any arbitrary segment of the key stream [12]. In this paper we will investigate high-speed Leviathan binary tree traversal architectures to support ordered and out-of-order key stream generation. Section 2 describes Leviathan stream cipher while Section 3 discusses parallel and pipeline architectures for binary tree based key stream generator. It describes the design of pipeline architecture. Section 0 presents the results for pipeline architecture implementation. Section 5 concludes this paper.
1.1 Related Research

Before designing architectures for a binary tree based key stream generator we will investigate a related application of tree traversal the routing table lookup within Internet routers. The routing table lookup is performed on arrival of each IP packet (i.e., ingress IP packet) by looking up the next-hop-address. Table 1 illustrates a typical routing table where the entries are of the form <route-prefix, next-hop-address> [14]. The route-prefix is matched with the significant bits of the 32-bit destination IP address of the ingress packet. The ingress packet is forwarded to the next-hop address associated with longest prefix matching with the significant bits of destination IP address. The longest prefix matching at high-speed motivates development of algorithms for searching the forwarding table to perform fast route lookups.

Table 1: Forwarding table with four prefixes and the corresponding next-hop addresses. A * denotes wildcard bits.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next-hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 111*</td>
<td>H1</td>
</tr>
<tr>
<td>P2 10*</td>
<td>H2</td>
</tr>
<tr>
<td>P3 1010*</td>
<td>H3</td>
</tr>
<tr>
<td>P4 10101*</td>
<td>H4</td>
</tr>
</tbody>
</table>

One of the commonly used approach for longest prefix match is to search a binary tree, commonly known as lookup trie, that has branches labeled ‘1’ or ‘0’. The ‘1’ and ‘0’ corresponds to the right and the left branches respectively. The binary tree is traversed performing search operation using significant bits of the destination IP address as search key. Traversing to particular node represents concatenating the labels of all the branches in the path from the root node. The longest prefix search operation proceeds bitwise starting from the root node of the lookup trie. The left (right) branch of the root node is taken if the first bit of the search key is ‘0’ (‘1’), following the remaining path of traversal is determined in a similar manner using successive bits of the search key. The search ends when left or right child pointers are pointing to a null pointer or does not have the matching bit. The concatenated labels create the longest prefix matching the significant bits of the destination address. A sample lookup trie is shown in Figure 2.

Figure 2: A lookup trie with nodes marked P1…P4 denote nodes which have valid next-hop-ptr. Concatenated branch labels along the path from the root node to the nodes P1…P4 implicitly form prefixes.

The search operation entails traversing the lookup trie. This is similar to the preorder traversal of the binary computation tree in Leviathan. The difference between the lookup trie and the computation tree prevents us from using the architectures used for searching the lookup trie for preorder traversal of binary computation tree. The differences are (a) the node state information (i.e., trie node structure) is stored permanently for the lookup trie in contrast it is only used once to compute the state information for the child node in the case of binary computation tree, (b) traversal order depends on the search key for lookup trie versus it is predetermined to preorder for computation tree, and (c) the result of comparison between the significant bits of the IP destination address and concatenation branch labels determine the level at which the
traversal of lookup trie stops versus all the leaf nodes of a complete binary tree are traversed in the case of binary computation tree.

2 Leviathan: A Binary Tree Based Stream Cipher

In Leviathan binary tree stream cipher the key stream is a concatenation of output keys computed at its leaf nodes. A key at a leaf node is obtained by computing functions at all nodes in the path from the root node. Leviathan is a complete binary tree based stream cipher parameterized by the number of bits n per output key word and the number of bits m in the user key. Preorder traversal of the binary computation tree generates an ordered key stream. In Leviathan there are \(2^{16}\) binary computation trees, each of height 16 and \(2^{32}\) leaf nodes. In this paper n is 32 bits, m is 64 bits, each output key is 32 bits and the input user key is 64 bits. Leviathan maps the 64-bit user key to \(2^{50}\) bytes (32 bits per key word \(2^{16}\) key words per binary tree \(2^{32}\) trees) of pseudorandom key stream.

![Figure 3: In a Leviathan computation tree of height 3, the function a (b) is executed when going from a node to its left (right) child. The function c is executed at the leaf nodes to generating the key stream.](image)

In a Leviathan tree of height 3 shown in Figure 3 nodes are labeled in a breadth-first-traversal fashion starting with the root node as 1. Associated with each node is a 96-bit state \(z | y | x\), where the z, y and x are each 32 bits. The state of the root node is \(1 | 0 | \text{tree #}\). The state at a node is used to obtain the state of its left and the right children using a (=f,d) and b (=g,d) functions respectively. Functions f, g and d are defined as:

\[
\begin{align*}
  f(z|y|x) &= 2z | S(R(S(R(y)))) | L(S(L(S(x)))) \\
  g(z|y|x) &= 2z+1 | L(S(L(S(y)))) | S(R(S(R(-x)))) \\
  d(z|y|x) &= z | x+y+z | 2x+y+z
\end{align*}
\]

R and L indicate right rotate and left rotate by 8-bits. S is a non-linear key dependent substitution function \(S(x_3|x_2|x_1|x_0)=x_3 \oplus S_3(x_0) | x_2 \oplus S_2(x_0) | x_1 \oplus S_1(x_0) | S_0(x_0)\), where tables \(S_0, S_1, S_2\) and \(S_3\) are setup using seed key. At a leaf node the output key word is computed by applying the function \(c(y|x)= x \oplus y\). An out-of-order key stream is generated by seeking to the particular leaf node followed by preorder traversal of the binary tree. High-speed generation of key stream is equivalent to high-speed preorder traversal of binary trees.

3 High-Speed Architectures for Binary Tree Based Key stream Generator

A straightforward approach to generating the 32-bit key words at all the leaf nodes of a binary tree of height h requires \(O(hx2^h)\) time for computing all functions and \(O(2^h)\) for storing all output key words. For a binary tree of height 16 this translates to \(O(16x2^{16})\) time for computing the functions a and b and \(2^{16}\) 32-bit memory locations for storing the key stream. Since the storage requirement is not practical the output key words should be generated dynamically. One approach to dynamically generating the key stream entails traversing the unique path from the root to the corresponding leaf node and computing the key words by executing the functions a or b at all the nodes on this path. This scheme yields \(O(hx2^h)\) time and \(O(1)\) storage. The time interval between consecutive output key words for this straightforward approach is \(T_{\text{key_period}} = 16xT\). Next, we will describe two high-speed tree traversal architectures that are significant improvements over this
3.1 Parallel Architecture

We propose an architecture wherein the state information computed at an intermediate node is stored and reused. In
this architecture the time interval between consecutive output key words is \((\lceil \frac{Nd}{2} \rceil + 1) \times T\) where \(Nd\) is the number of
nodes between consecutive leaf nodes in an inorder traversal of the binary tree. The time interval between consecutive keys
is proportional to the number of nodes between the corresponding leaf nodes. As shown in Figure 4 (a) the time interval
between output key words generated by nodes 10 and 11 is different from the time interval between output key words
generated by nodes 11 and 12. This is because there is one node (node 5) between nodes 10 and 11 and five nodes (nodes 5,
2, 1, 3 and 6) between nodes 11 and 12. Figure 4 (b) summarizes the time intervals between consecutive key words.

![Figure 4](image)

**Figure 4:** (a) Time interval between consecutive keys is \(\propto\) to the number of nodes between the corresponding leaf nodes. (b) \(T_{key\_period}\) for consecutive leaf node pairs for a binary tree of height 3

Computing key words at consecutive leaf nodes in parallel can bound the variation in \(T_{key\_period}\). Specifically, if
\(T_{key\_period}\) between consecutive key words exceeds a predetermined bound, the paths leading to the consecutive output key
words are computed in parallel. For example, consider a predetermined bound of 3 for the binary tree of height 3 in Figure
5. During ordered key stream generation, node 5 along the path to the output key word at node 11 and node 1 along the path
to the output key word at node 12 are carried out in parallel. Similarly, during out-of-order key stream generation of key
words corresponding to nodes 11 and 12, the functions a and b at node 1 along the paths to these nodes are carried out in
parallel.

![Figure 5](image)

**Figure 5:** Illustrates functions traversed in parallel for generating (a) ordered and (b) out-of-order output keystream.

From Figure 5 (a) and (b) it can be seen that the condition for starting parallel traversal is different for computing
ordered and out-of-order output key words. For the ordered output key words the parallel traversal is initiated if the
intermediate node visited is root of a sub-tree with the rightmost leaf node greater than the threshold \(Nd\) apart from the next
leaf. For the out-of-order output key words the parallel traversal is initiated if an intermediate node visited is \(\lceil \frac{Nd}{2} \rceil\) nodes
apart from the leaf whose state information is used to compute the output key word being seek.
Preorder traversal of the binary computation tree entails visiting nodes using a “node-left sub tree-right sub tree” rule. When a node is first visited both the a and b functions are computed. While the 96-bit result of the a function is used immediately, the result of b function is stored for use when the right sub tree of the node is visited. The first column of Table 2 lists the order in which functions a and b are computed, stored and used during the preorder traversal of binary tree of height 3. For computing output key word at node 8, the functions a and b at intermediate nodes 1, 2 and 4 are computed entailing 3 additional storage elements for the state information of nodes 3, 5 and 9. The total additional storage for binary tree equals storage element bits x tree height, giving 96 x 3 bits for the tree of height 3. The total additional storage for binary computation tree of height 16 would equal 96 x 16 bits.

Let us look at when the results of functions a and b are used. While the results of function a are used when a node is first visited, the results of function b are stored when a node is first visited and used later. In order to reduce the storage requirement for the intermediate node state information, we propose to traverse the binary computation tree using an intertwined preorder, inorder traversal which entails visiting nodes using the Node\textsubscript{a}-Left-Node\textsubscript{b}-Right rule. The Node\textsubscript{a} and Node\textsubscript{b} represent computing functions a and b. The preorder, inorder traversal essentially superimposes the preorder traversal of functions a and the inorder traversal of functions b. For the intertwined preorder, inorder traversal the intermediate nodes are visited in the order of 1\textsubscript{a}, 2\textsubscript{a}, 4\textsubscript{a}, 8\textsubscript{c}, 4\textsubscript{b}, 9\textsubscript{c}, 2\textsubscript{b}, 5\textsubscript{a}, 10\textsubscript{c}, 5\textsubscript{b}, 11\textsubscript{c}, 1\textsubscript{b}, 3\textsubscript{a}, 6\textsubscript{a}, 12\textsubscript{c}, 6\textsubscript{b}, 13\textsubscript{c}, 3\textsubscript{b}, 7\textsubscript{a}, 14\textsubscript{c}, 7\textsubscript{b}, 15\textsubscript{c}. If the assumption is made that the functions at leaf node are computed with functions at their parent node, the combined computation written as single operation gives the node sequence of 1\textsubscript{a}, 2\textsubscript{a}, (4\textsubscript{a}, 8\textsubscript{c}), (4\textsubscript{b}, 9\textsubscript{c}), 2\textsubscript{b}, (5\textsubscript{a}, 10\textsubscript{c}), (5\textsubscript{b}, 11\textsubscript{c}), 1\textsubscript{b}, 3\textsubscript{a}, (6\textsubscript{a}, 12\textsubscript{c}), (6\textsubscript{b}, 13\textsubscript{c}), 3\textsubscript{b}, (7\textsubscript{a}, 14\textsubscript{c}), (7\textsubscript{b}, 15\textsubscript{c}) for the binary tree of height 3. The functions b are computed in an inorder sequence of 4\textsubscript{b}, 2\textsubscript{b}, 5\textsubscript{b}, 1\textsubscript{b}, 6\textsubscript{b}, 3\textsubscript{b}, 7\textsubscript{b} and the functions a are computed in an preorder sequence of 1\textsubscript{a}, 2\textsubscript{a}, 4\textsubscript{a}, 5\textsubscript{a}, 3\textsubscript{a}, 6\textsubscript{a}, 7\textsubscript{a}. The right column of Table 2 lists the order of functions computed with preorder, inorder traversal of binary tree of height 3, in which the state information for sibling nodes is not stored. The state information for nodes 8 and 9 use the same storage elements in the intertwined preorder, inorder traversal given identical resources.

Table 2: The function computation order for preorder and preorder, inorder traversals of the binary tree height 3 and storage element requirements.

<table>
<thead>
<tr>
<th>preorder traversal</th>
<th>intertwined preorder, inorder traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a at node 1</td>
<td>a at node 1</td>
</tr>
<tr>
<td>a at node 2</td>
<td>a at node 4</td>
</tr>
<tr>
<td>b at node 2, store b</td>
<td>c at node 8</td>
</tr>
<tr>
<td>a at node 4</td>
<td>b at node 4</td>
</tr>
<tr>
<td>b at node 4, store b</td>
<td>c at node 9</td>
</tr>
<tr>
<td>c at node 8</td>
<td>b at node 2</td>
</tr>
<tr>
<td>c at node 9, empty b at node 4</td>
<td>a at node 5</td>
</tr>
<tr>
<td>a at node 5, empty b at node 2</td>
<td>c at node 10</td>
</tr>
<tr>
<td>b at node 5, store b</td>
<td>b at node 5</td>
</tr>
<tr>
<td>c at node 10</td>
<td>c at node 11</td>
</tr>
<tr>
<td>c at node 11, empty b at node 5</td>
<td>b at node 1</td>
</tr>
<tr>
<td>a at node 3, empty b at node 1</td>
<td>a at node 3</td>
</tr>
<tr>
<td>b at node 3, store b</td>
<td>a at node 6</td>
</tr>
<tr>
<td>a at node 6</td>
<td>c at node 12</td>
</tr>
<tr>
<td>b at node 6, store b</td>
<td>b at node 6</td>
</tr>
<tr>
<td>c at node 12</td>
<td>c at node 13</td>
</tr>
<tr>
<td>c at node 13, empty b at node 6</td>
<td>b at node 3</td>
</tr>
<tr>
<td>a at node 7, empty b at node 3</td>
<td>a at node 7</td>
</tr>
<tr>
<td>b at node 7, store b</td>
<td>c at node 14</td>
</tr>
<tr>
<td>c at node 14</td>
<td>b at node 7</td>
</tr>
<tr>
<td>c at node 15, empty b at node 7</td>
<td>c at node 15</td>
</tr>
</tbody>
</table>

3.1.1 Extension to m-ary tree traversal
The parallel traversal architecture for the binary tree based keystream generator can be extended to m-ary tree based
keystream generator, where \( m = 2, 4, 8 \ldots \). The Figure 6 illustrates an \( m \)-ary tree where \( m = 4 \) and tree height = 2. The parallel traversal approach for binary tree can be extended to traverse \( m \)-ary tree because the assumption that the branches traversed in parallel have identical computation times, \( T = \max(T_{a1}, T_{b1}, \ldots, T_{an}, T_{b2}, T_{b3}) \), where \( T \) is the maximum of all the times to compute any of the functions at intermediate nodes and leaf nodes, can be extended for \( m \)-ary tree. Also the preorder-inorder traversal rule of Node\(_{a1}\)-Left…Node\(_{an}\)-Left-Node\(_{b1}\)-Right…Node\(_{bn}\)-Right is extended for traversal of nodes in an \( m \)-ary tree, where Node\(_{a1}\)…Node\(_{an}\) and Node\(_{b1}\)…Node\(_{bn}\) represent computing functions \( a1…an \) and \( b1…bn \) respectively. If an \( m \)-ary tree is preorder traversed the total additional storage elements required for node state information would equal storage element bits \( \times \) tree height \( \times m-1 \). The time period between consecutive output key words: \( T_{\text{key-period}} = \left\lceil \frac{N_d}{2} \right\rceil + 1 \times T \). The conditions to start the parallel traversal for the ordered and out-of-order output key words is same as for binary tree traversal given that sub-trees computed in parallel balance the times taken to compute all their functions.

![Figure 6: 4-ary tree of height 2 where the leaf numbers are 4 bits wide.](image)

3.2 Pipeline Architecture

Overlapping the computation of functions along paths leading to the consecutive output key words generate keystream at uniform rate, independent of number of nodes between the consecutive leaf nodes. The time period between computing the consecutive output key words \( T_{\text{key-period}} \) is equal to the time to compute the function at leaf node: \( T_{\text{key-period}} = T_c \), when number of paths overlapping is equivalent to height of the tree. The overlapping computation of functions along the paths leading to the consecutive functions at leaf nodes essentially hides the latency entailed in computing the consecutive output key words.

![Figure 7: Pipeline traversal of computation tree of height 3. Function cycle = 3 clock cycles.](image)
In pipeline architecture the computation of functions in the path to the current and the consecutive output key words are pipelined. The depth of the pipeline is equal to the height of the binary computation tree plus one for an additional stage for function at leaf node. An assumption is made that all the functions computed in pipeline stages have identical computation times: \( T = \max(T_a, T_b, T_c) \), where \( T \) is the maximum of all the times to compute any of the functions at intermediate nodes and leaf nodes. Figure 7 shows pipelined paths for the binary tree of height 3, where the functions at nodes 1, 2, 4, 8, 1, 2, 4, 9 and so on are computed in that order.

The functions along the path traversed from the root node to the leaf node are computed in successive stages of the pipeline and their results are stored in the stage registers. The results from the pipeline stage registers are used by the following stages to compute the state information of the node at the next level of the tree. As shown in Table 3, the second iteration for function \( a \) at node 1 overlaps with the first iteration for function \( a \) at node 2. Here DSXSRY denotes compute function \( a \rightarrow \) read from ROM location specified by the least significant 8 bits of \( X \) (SX) and read from ROM location specified by the least significant 8 bits of the 8-bit right rotated \( Y \) (SRY) \( \rightarrow \) in next clock cycle XOR the resulting data read from the ROM with its respective addresses.

Table 3: Illustrates sample cycle-by-cycle operations within the 4-stage pipeline when traversing a binary tree of height 3 in preorder. D: compute function \( d \), S: read from ROM location specified by least significant 8 bit of the 32-bit argument and XOR the data read from ROM with the address in next clock cycle, R: right rotate 32-bit argument by 8 bits, L: left rotate 32-bit argument by 8 bits, C: function at leaf node, X: 32-bit argument, Y: 32-bit argument, N: complement of 32-bit argument and P: pass 96-bit state information to next stage by writing it to the pipeline stage register.

<table>
<thead>
<tr>
<th>Clock #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 1</td>
<td>DSX SRY</td>
<td>SLX SRY</td>
<td>LXP</td>
<td>DSX SRY</td>
<td>SLXS RY</td>
<td>LXP</td>
<td>DSX SRY</td>
<td>SLX SRY</td>
<td>LXP</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word 2</td>
<td>DSX SRY</td>
<td>SLXS RY</td>
<td>LXP</td>
<td>DSX SRY</td>
<td>SLX SRY</td>
<td>LXP</td>
<td>DSR NXS Y</td>
<td>SRX SLY</td>
<td>LYP</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the tree of height 3, overall 12 clock cycles are required to generate each of the 32-bit output key word. The 12 clock cycles are attributed to 3 clock cycles per stage for executing function plus an additional 3 clock cycles for the stage executing function at leaf node. For Leviathan binary computation tree of height 16, overall 51 clock cycles are required to generate each of the 32-bit output key words.

Figure 8: Pipeline architecture for 16-stage keystream pipeline plus an additional stage for function at leaf node. SBOX: substitution table implemented as RAMs/ROMs. In the case key setup is performed on general purpose processor the SBOX can be implemented as ROMs.
Figure 8 shows the design of pipeline architecture computing Leviathan keystream divided into three blocks, the key setup, the pipeline and the pipeline controller. The keystream pipeline computes keystream by implementing function at intermediate node in its successive stages. For 4 stage pipeline, during first 3 clock cycles stage 0 would execute function a at node 1 specified by operation sequence DSXSRYÆSLXSRYÆLXP along path to output key word at leaf node 8. Successively during second 3 clock cycles stage 0 would execute second iteration of function a at node 1 along the path to output key word at leaf node 9 and stage 1 would execute function a at node 2 along path to output key word at node 8.

The pipeline stages compute sub-functions d, f or g of functions a or b respectively, taking overall 3 clock cycles. The 3 clock cycles are attributed to the reads from substitution table which take 2 clock cycles and to the pipeline stage registers which add 1 clock cycle delay. Within the pipeline stages, the intermediate values of X, Y and Z are stored in temporary registers at every clock cycles. For function a, specified by operation sequence DSXSRYÆSLXSRYÆLXP, the intermediate values stored in the pipeline stage’s internal registers at the end of first clock cycle are X and 8-bit right rotated Y, at the end of second clock cycle the 8-bit left rotated XORed X and 8-bit right rotated XORed Y and at the end of third clock cycle the 8-bit left rotated XORed X and XORed Y. The XORed operation is with respect to the ROM address of the X and Y arguments.

3.2.1 Keystream Pipeline

The pipeline architecture was selected for implementation as it computes keystream at uniform rate: \( T_{\text{key\_period}} = T_c \).

For Leviathan keystream pipeline shown in Figure 8, there are 16 pipeline stage registers to store the node state information and functions at intermediate and leaf nodes implemented as pipeline stages with 2 256x32 substitution table ROMs associated with each stage. The controller specifies stage-by-stage behavior of the pipeline via \( function\_select \) control signal. All pipeline stage implementations, except the one implementing the function at leaf node, are identical.

Each stage of the pipeline implements functions a and b using 32-bit additions, bit-wise complement and rotate operators. The 32-bit entries of substitution table get written by the key setup block and are read by the pipeline stage block implementing functions a and b. The pipeline stages implement two substitution tables enabling the computation of components of sub-functions f or g in parallel. For function f, defined as \( f(z|y|x)= 2z | S(R(S(R(y)))) | L(S(L(S(x)))) \), computation of y and x components are performed in parallel given two copies of substitution table.

3.2.2 Pipeline Controller

The controller via the \( function\_select \) signal feed to each of the pipeline stages indicate, which of the two functions a or b is to be executed in the next clock cycle. The pattern of \( function\_select \) signals values indicate the preorder traversal of the binary computation tree. The data flow in the pipeline is node state information \( z | y | x \) resulting from execution of functions a or b implemented in pipeline stage. The execution of function a or b in pipeline stages result in node state information input to successive stage which executes the function a at next level node of the binary tree. The leaf numbers specify the path from the root to particular leaf node. The preorder traversal of binary tree of height 16 starts from the root traversing first to the left most leaf following path \( \{0\}^{16} \). For generating out-of-order output key word the starting leaf number (i.e., the path to the leaf being seek) could be provided as an input, indicating the seek operation. The \( function\_select \) signal gets new value every 3 clock cycles, the number of clock cycles required for computing the functions plus the stage register delay.

Figure 9 shows the controller architecture for binary tree of height three. It is designed using one 3-bit counter, three 3-bit registers, inverters and muxs. The 3-bit counter is labeled as \( leaf\_counter \), three 3-bit registers are labeled as \( path\_registers \) and muxs are labeled \( path\_mux, bit\_mux, and stage\_mux \). The state of the controller and mux inputs are set upon \( done \) active. The \( done \) becomes active every three clock cycles. The \( function\_select \) signal is constructed by reading the path register bits from the most significant bit to the least significant bit positions of the path registers. The path register bit position read changes upon \( done \) active. The most significant bit is read when the leaf number is inserted into particular
path register, the second most significant bit is read upon next done active and so on. The reading of least significant bit indicates completion of traversal to particular leaf number stored in the path register. After reading the least significant bit from the path register, a new leaf number is inserted into that particular path register on next done active and the process of traversal to newly inserted leaf is restarted by reading the most significant bit. The path to the leaf node already traversed is discarded upon done active. The leaf counter increments by one upon done active. Also upon done active one of the path registers is feed the path_mux output and the bit position of path registers are feed as bit_mux inputs. Path registers control function executing in particular pipeline stage and is updated every 3xh clock cycles upon done active by selecting inputs to the stage_mux.

The binary tree of height 3 has 8 leaf nodes. Figure 9 shows the controller state before and after the insertion of path 011 for leaf number 3 in the path register. Upon insertion of the leaf number 3, path registers would have values <001>, <010>, <011>, which is different from the path register values <000>, <001>, <010>. In contrast to the preorder traversal, a stack structure is not used to track the nodes visited with respect to the root. Instead the presented controller uses leaf number as path from the root node to indicate the branches traversed.

**Figure 9**: Controller architecture for height 3 computation tree. Traversal starts from leaf number 000. The path register whose last significant bit has been read ← leaf counter and leaf counter ← leaf counter + 1 upon done active.

### 3.2.3 Discussion

The key setup block was implemented and its results are presented, though no detailed architecture study was done. The key setup computes four 256x8 tables that store invertible substitution functions, S₀, S₁, S₂ and S₃. These tables are generated pseudo-randomly by swapping elements at random in seed key dependent manner. These four substitution tables, also known as SBox, are generated every 2¹⁰ bytes of output key words using the 8 byte seed key and read by the pipeline stage functions. The key setup is not in the critical path for generating the keystream, therefore key setup could be implemented on a general purpose processor. The key setup design consists of 32-bit registers, 16-bit counters, 32-bit adders and mod operators. All four 256x8 tables are concatenated into single 256x32 substitution table, reducing number of reads issued by the pipeline stage functions from four to one. This is possible as all the sub-components of the function S index to the same location in RAM/ROM. S function is implemented as RAM/ROM lookup (read) issued from the pipeline stage function.

The pipeline traversal can be optimized further at increased storage requirements. Traversals of the overlapping paths can avoid re-computing the functions at intermediate node that have already been computed while traversing to the previous output key words. The additional storage is entailed in tracking the paths already traversed. Table 4 illustrates the path table that store distinct set of node numbers along the paths already traversed for a binary tree of height 3.
Table 4: Path Table stores the paths traversed as distinct set of nodes

<table>
<thead>
<tr>
<th>leaf node number</th>
<th>node numbers</th>
<th>align path</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>&lt;1, 2, 4&gt;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&lt;1, 2, 5&gt;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>&lt;1, 3, 6&gt;</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>&lt;1, 3, 7&gt;</td>
<td></td>
</tr>
</tbody>
</table>

3.2.4 Extension to m-ary tree

The pipeline architecture for traversal of binary tree can be extended to traversal of m-ary tree of height h. The differences between the binary tree and the m-ary tree implementations are in the pipeline stage function and the controller. The pipeline stage implements m different functions instead of 2 functions of binary tree pipeline stage. The controller for m-ary tree traversal can use the leaf numbers just as in binary tree to construct the function select signal. The function select signal to each stage would be q-bits wide, where m = 2^q. For example, 4-ary tree would require the function select signal to be 2-bit wide instead of single bit as is the case for implementation of binary tree based keystream pipeline stage. The similarities between the pipeline architecture for the binary tree and the m-ary tree design are: (a) fixed period between computing consecutive output key words \( T_{key_period} = T_r \) (b) the depth of the pipeline is equal to the height of the m-ary computation tree plus an additional stage for the function at leaf node and (c) the functions along the paths of m-ary tree traversed from the root node to the leaf node are computed in successive stages of the pipeline and their results are stored in the stage registers used by the following stages to compute the state information of the node at the next level. The pipeline traversal to first 8 leaf nodes of 4-ary tree of height 2 is illustrated in Figure 10.

![Figure 10: Pipeline traversal of 4-ary tree of height 2. Function cycle = 3 clock cycles.](image)

Table 5 summarizes the order of time taken to compute functions at intermediate nodes and storage requirement for the traversal architectures presented for m-ary computation tree.

Table 5: Traversal architectures complexity

<table>
<thead>
<tr>
<th>Approach</th>
<th>Computation Time</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straightforward</td>
<td>( O(hm^h) )</td>
<td>( O(m^h) )</td>
</tr>
<tr>
<td>Parallel</td>
<td>( O(m^h) )</td>
<td>( O(Nd) )</td>
</tr>
<tr>
<td>Pipeline</td>
<td>( O(h) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>
4 Results

In order to measure the throughput of pipeline we evaluate the key setup and pipeline block separately. The key setup can be executed concurrently while computing the keystream in practical systems. Implementation results are obtained by targeting Xilinx Virtex II [15] series FPGAs. The ModelSim SE [16] and the Synplify Pro [17] are used to perform functional simulation. The design guidelines followed are in [18]. The timing simulation of controller and pipeline blocks are performed using the timesim.vhd to verify against the functional simulation. The timesim.vhd file was generated by Xilinx ISE 5.1i. The post place and route results from Xilinx ISE 5.1i of the pipeline and its controller blocks give the area = 6864 slices, frequency = 50 MHz and throughput = 500 Mbps. The clock period was rounded upwards from 19.95 ns to 20 ns, obtained from the Timing Analyzer of Xilinx ISE tool which analyzes post place and route static timing.

The waveform snapshots presented are from functional simulation using the clock period obtained from Timing Analyzer. Figure 11 shows the simulation snapshot of interaction between the controller and the pipeline. The signal marked 1 point to the function_select signal when the 15th pipeline stage is computing the leaf node state information used to compute the 1st output key word. The signals marked 2, 3 and 4 shows the output of the 15th pipeline stage used in computing the 1st output key word and the function_select signal used to compute the 2nd output key word. Signals marked 5, 6 and 7 repeat same actions as described by signals marked 2, 3 and 4 for computing 2nd output key word and the function_select signal used for computing 3rd output key word.

Figure 11: Simulation waveform of the interaction between the controller and the pipeline stages.

Figure 12 shows the simulation snapshot of the 15th pipeline stage when computing function f. The signals marked show the state of pipeline stage function in each clock cycle. The signal marked 1 indicates the start_funcs signal indicating start of the pipeline stage function. The start_funcs signal stays active only for single clock cycle. The signal marked 2, 3 and 4 point to output from previous pipeline stage, read addresses issued to ROM1 and ROM2, and update of temporary registers to store intermediate values of x, y and z respectively. In cycle 2 the signals marked 5 and 6 repeat actions for signals marked by 3 and 4 for successive stage. At the end of 3rd clock cycle signal marked 7 shows the state information for the next node as value stored in pipeline stage register.
Figure 12: Simulation waveforms of the pipeline stage 15 executing function a.

5 Conclusion

From various architectures we chose to implement the pipeline architecture because it offers uniform rate for computing keystream. In comparison to software implementation on Pentium II with clock rate of 500 MHz on custom ASIC design that achieves throughput of 375 Mbps [11], the simple pipeline architecture design with clock rate of 50 MHz on merchant silicon FPGAs yields throughput of 500 Mbps. The pipeline architecture is flexible and can easily be modified to increase the throughput. The throughput for computing the keystream using pipeline architecture can be nearly doubled (~1 Gbps) if the pipeline stage 15 computes functions a and b in parallel and instantiating a second copy of the function c. These changes will yield 64-bits instead of 32-bits of keystream every 3 clock cycle, effectively doubling the throughput at increase in hardware. If the design was implemented on ASIC technology it would yield better results.
6 References