An FPGA Design of a Unified Hash Engine for IPSec Authentication

Esam Khan, M. Watheq El-Kharashi, Fayez Gebali, and Mostafa Abd-El-Barr
Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada
Emails: {ekhan, watheq, fayez, mabdelba}@ece.uvic.ca

Abstract—Hash functions are important security primitives used for authentication and data integrity. Among the most popular hash functions are MD5, SHA-1, and RIPEMD-160 that are used in conjunction with HMAC for IPSec. These three hash functions are based on an older one, MD4. Therefore, they have some similarities that can be exploited for designing a unified engine to perform the three hash functions. A unified engine design proves useful since the three algorithms are to be used by the same implementation on the same core for authentication and data integrity using HMAC for IPSec. In this work, we design a SoC with a unified hash engine that can be reconfigured at runtime to perform one of the three hash functions. The results of our work show that the proposed engine has a balance between area and throughput compared to previous works.

I. INTRODUCTION

Cryptographic Hash Functions are important security primitives that are used in different applications. The most common ones include data integrity and authentication [1]. There exists a number of hash functions that are used in cryptography. Among the most popular ones are the three children of an older hash function, MD4; namely, MD5 [2], SHA-1 [3], and RIPEMD-160 [4]. These functions can also be combined with some encryption methods to enhance security. One of the standard algorithms used with hash functions is HMAC (The Keyed-Hash Message Authentication Code) [5]. The resulting algorithms then became HMAC-MD5 [6], HMAC-RIPEMD-160 [7], and HMAC-SHA-1 [8]. These three algorithms are the standard authentication algorithms used for IPSec, as shown in Fig. 1 [9].

Fig. 1 illustrates that IPSec highly uses HMAC with the three hash functions since both ESP and AH are using these algorithms alternatively for authentication and data integrity. Therefore, it is important to have the three hash functions available for any implementation of IPSec. However, it is costly to have three modules for the three algorithms on the same hardware core. Since the three hash functions are based on the same algorithm, MD4, they have a number of similarities that may be exploited to design a unified engine that is reconfigurable. In this work, we show a design of an engine that exploits the similarities of the three hash functions to save area, time, and power consumption. This engine can be used with HMAC for IPSec or with any other method that uses hash functions such as digital signature.

The paper is organized as follows. In Section II, a description of the three hash functions and their similarities and differences are discussed. The proposed architecture and how it exploits the similarities of the hash functions is shown in Section III. Then, we discuss the simulation results and compare them to published results in Section IV. Finally, we conclude the paper in Section V.

II. DESCRIPTION OF MD4-BASED HASH FUNCTIONS

The general skeleton of the three hash functions consists of the following steps [1]:

1) Initialization: In this step, some constants are defined. These constants include initial chaining values (IVs), denoted by \( c \), order of accessing message words, denoted by \( z \), additive constants, denoted by \( y \), and the number of bits for left-rotation in each step, denoted by \( s \).

2) Preprocessing: The message to be hashed has to be of length divisible by 512. Therefore, the message is appended with a single bit of value ‘1’, followed by the required number of 0’s to make the message length 64 bits less than a multiple of 512. The last 64 bits represent the original message length. After padding, the message is split into 512-bit blocks, each of which consists of 16 32-bit words. In addition, a number of chaining variables are initialized in this step.

3) Processing: This is the heart of the algorithms, where each 512-bit block is processed in a step. Each step consists of the following substeps:
   - Initialize working variables with the current values of the chaining variables.
• Update the working variables using some computation in rounds. Each round has the same computation in its all steps.
• Update the chaining variables

We will elaborate on this step when we talk about the proposed architecture.

4) Completion: The final hash value is composed by appending the chaining variables.

This simplified skeleton shows how similar these algorithms are. Details of each algorithm are presented in [1]–[4]. Although the three hash functions have similarities, there are some differences that must be taken into consideration when designing a unified engine. These differences can be summarized in the following:

• The representation of the message words and the final hash value must be in little-endian for RIPEMD-160 and MD5, whereas they are represented in big-endian for SHA-1.
• The 16 32-bit message words are expanded to 80 words in SHA-1. Then, in each compression step, one word is referenced in a serial fashion. However, RIPEMD-160 and MD5 use only the 16 words, but in different accessing orders.
• For each 512-bit message block, MD5 has 4 rounds, each consists of 16 steps, SHA-1 has 4 rounds, each consists of 20 steps, and RIPEMD-160 has 5 rounds, each consists of 16 steps.
• Working variables are not updated in the same way in compression steps.
• RIPEMD-160 uses two parallel processing blocks for each 512-bit message block, whereas there is only one processing block used in the other two.
• After processing each 512-bit message block, MD5 and SHA-1 update the chaining variables in the same way, whereas RIPEMD-160 uses a different way to update them.
• The final hash value of MD5 is 128 bit long, whereas it is 160 bit long for the other two.

III. PROPOSED ARCHITECTURE

Based on the similarities and differences of the three hash functions, we propose a unified algorithm and then implement this algorithm on a unified architecture. The unified algorithm consists of two blocks, a main block and a processing block. The main block is responsible for the initialization, preprocessing, and completion parts as described in the general skeleton above. On the other hand, the processing block performs the processing part of the unified algorithm.

Fig. 2 shows the main block of the unified algorithm. This algorithm produces a 160-bit hash, and if the required hash function is MD5, the most significant 128 bits are selected.

We notice that in this block, the only difference is in the little-endian representation for MD5 and RIPEMD-160. On the other hand, more work is needed for designing a unified processing block (Fig. 3).

In the processing block shown in Fig. 3, an expansion block is included for SHA-1. This step expands the 16 words of the message block under processing, \(X(0)\) to \(X(15)\) into 80 words. Then, the working variables \(A\) to \(E\) are initialized with the values of the chaining variables \(H1\) to \(H5\), respectively. Following this, 80 rounds of processing are performed. However, if the algorithm is MD5, the processing stops after 64 rounds. In each round, a temporary variable \(T\) is computed. This is done using the function \(Compute\_T\), which is the addition modulo 2\(^{32}\) of some arguments. These arguments differ from a round to another depending on the algorithm. One of these arguments is a round function \(f\). In the original hash algorithms, every round in each algorithm has a different function. In our unified engine, we compressed these functions into four functions, \(f1, f2, f3,\) and \(f4\), such that:

\[
\begin{align*}
\text{f1}(u, v, w) &= (u \land v) \lor (\lnot(u) \land w), \\
\text{f2}(u, v, w) &= (u \land v) \lor (u \land w) \lor (v \land w), \\
\text{f3}(u, v, w) &= u \lor v \land w, \text{ and} \quad f4(u, v, w) = u \land (v \land \lnot w),
\end{align*}
\]

where \(u, v,\) and \(w\) are the three inputs to the round function in that order. After computing \(T\), the working variables are updated.

As we mentioned in Section II, the three hash functions differ in the number of rounds and the number of steps per round. In order to combine the three algorithms, the number of steps in each round has to be modified. This modification resulted in 7 rounds for the three hashing methods, and one additional round for SHA-1 and RIPEMD-160.

For RIPEMD-160, a parallel processing block is required. This block is only activated if the required algorithm is RIPEMD-160. This saves power consumption for the other two. This block has the same processing steps as the main RIPEMD-160 processing, but using different working vari-
The final step in each processing iteration is the update of the chaining variables. For SHA-1 and MD5, it is simply done by adding the values of the working variables \( A \) to \( E \) to the current value of \( H1 \) to \( H5 \), respectively. For RIPEMD-160, it is more complicated as shown in the figure.

Although the algorithm is shown serially in the figure, we could utilize some parallelism available to reduce the time and/or area required. This utilization can be summarized in the following:

1) Splitting the message block into 16 words can be done in parallel with the initialization of the working variables. 
2) Since SHA-1 accesses the message words serially, we could use only the original 16 words, and then perform the expansion process in parallel with the processing. Each word \( X(j) \) will be stored in \( X(j \ mod\ 16) \) after SHA-1 has already accessed \( X(j \ mod\ 16) \).

3) The parallel processing block of RIPEMD-160 can be run in parallel with the main processing block.

IV. SIMULATION RESULTS

In the implementation of the proposed architecture, we used Handel-C [10], which enables a high-level description with low-level manipulation. This language, along with its compiler DK, provided by Celoxica Ltd. [11], can be used to exploit parallelism, pipelining, and logic optimization. We tried to balance area and throughput, which depends on the maximum frequency and the number of cycles required to perform the required task.

We simulated our design and built it for Xilinx Vertex II FPGA (XC2V3000), which has 3584 CLBs. The simulation results are summarized in Table I.

<table>
<thead>
<tr>
<th>LUTs usage</th>
<th>40.57%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum frequency</td>
<td>37.0 MHz</td>
</tr>
<tr>
<td>Estimated power consumption</td>
<td>637 mW</td>
</tr>
<tr>
<td>Clock cycles for one 512-bit block for MD5</td>
<td>130</td>
</tr>
<tr>
<td>Clock cycles for one 512-bit block for SHA-1</td>
<td>162</td>
</tr>
<tr>
<td>Clock cycles for one 512-bit block for RIPEMD-160</td>
<td>162</td>
</tr>
<tr>
<td>Average MD5 throughput of a 512-bit block</td>
<td>145.72 Mbps</td>
</tr>
<tr>
<td>Average SHA-1 throughput of a 512-bit block</td>
<td>116.94 Mbps</td>
</tr>
<tr>
<td>Average RIPEMD-160 throughput of a 512-bit block</td>
<td>116.94 Mbps</td>
</tr>
</tbody>
</table>

The LUT usage of our design covers 1454 CLBs (40.57% of the available logic) including the memory cells required for the process. This includes both the main block and the processing block.

Table II shows the results of our proposal in comparison with previous works. We only considered works that integrated more than one hash algorithm. The work in [12] integrates MD5 and RIPEMD-160. The work in [13] combines SHA-1 with MD5. In [14], SHA-1 and MD5 are integrated with another hash algorithm, HAS-160 (The Hash function Algorithm Standard). MD5, RIPEMD-160, SHA-1, and SHA-256 are integrated in [15]. In [16], SHA-1 and RIPEMD-160 are integrated.

Most of these works reported figures only for the processing block. However, due to the need for the preprocessing block, memory, and routing resources, we include their area cost in our figures. This justifies the higher area cost that we are reporting. In addition, having a unified engine for the three algorithms is still better in area than having a module for each one of them.

On the other hand, when we look at the throughput, we find that our work is better than those integrating three or more algorithms [14, 15]. We also achieve better throughput than the work in [12] for RIPEMD-160, but less for MD5. However, SHA-1 is not included in that work. Although our throughput is slightly less than that achieved by the work
TABLE II
Comparison with previous works.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA vendor</td>
<td>Altera</td>
<td>Altera</td>
<td>Altera</td>
<td>Xilinx</td>
<td>Xilinx</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Area cost**</td>
<td>1964 LCs</td>
<td>3040 LCs</td>
<td>10573 LEs</td>
<td>1004 CLBs</td>
<td>2245 CLBs</td>
<td>1454 CLBs</td>
</tr>
<tr>
<td>Average frequency (MHz)</td>
<td>26.66</td>
<td>22.67</td>
<td>18.0</td>
<td>42.9</td>
<td>55.0</td>
<td>37.0</td>
</tr>
<tr>
<td>Clock cycles for MD5****</td>
<td>66</td>
<td>65</td>
<td>65</td>
<td>206</td>
<td>-</td>
<td>130</td>
</tr>
<tr>
<td>Average MD5 throughput (Mbps)***</td>
<td>206</td>
<td>178.6</td>
<td>142</td>
<td>107</td>
<td>-</td>
<td>145.72</td>
</tr>
<tr>
<td>Clock cycles for SHA-1****</td>
<td>-</td>
<td>81</td>
<td>81</td>
<td>255</td>
<td>20+1</td>
<td>162</td>
</tr>
<tr>
<td>Average SHA-1 throughput (Mbps)***</td>
<td>-</td>
<td>143.3</td>
<td>114</td>
<td>86</td>
<td>1339</td>
<td>116.94</td>
</tr>
<tr>
<td>Clock cycles for RIPEMD-160****</td>
<td>162</td>
<td>-</td>
<td>-</td>
<td>337</td>
<td>16+1</td>
<td>162</td>
</tr>
<tr>
<td>Average RIPEMD-160 throughput (Mbps)***</td>
<td>84</td>
<td>-</td>
<td>-</td>
<td>65</td>
<td>1656</td>
<td>116.94</td>
</tr>
</tbody>
</table>

* We only include results for the three hash functions MD5 (M), SHA-1 (S), and RIPEMD-160 (R)
** LC = Logic Cell, LE = Logic Element, CLB = Configurable Logic Block
*** Per one 512-bit block

in [13] for both MD5 and SHA-1, but that work did not include RIPEMD-160, which is more complicated in terms of area and processing requirements. In [16], SHA-1 and RIPEMD-160 are integrated in a replicated, pipelined fashion, and hence the highest throughput was achieved for those two algorithms. However, MD5 is not included in that work as well as the area consumption is much more than all other works, including ours.

We also include the estimated power consumption of our engine, as shown in Table I. However, because the other works did not include any criteria that help finding an estimated value for power consumption, we could not compare with them.

V. CONCLUSION

In this paper, we have designed a unified engine to perform hashing using MD5, SHA1, and RIPEMD160. We implemented this engine using Handel-C and synthesized it into a Xilinx Vertex II FPGA. The results obtained show that the proposed architecture is promising and could be used to enhance the timing, area, power consumption of the hashing process for security applications.

Future work includes integrating an HMAC block to the proposed engine to be used for authentication and other security tasks for IPSec in particular.

REFERENCES