FPGA Implementation of an HMAC Processor based on the SHA-2 Family of Hash Functions

Marcio Juliato
Dept. of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada, N2L 3G1
mrjuliat@uwaterloo.ca

Catherine Gebotys
Dept. of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada, N2L 3G1
cgebotys@uwaterloo.ca

ABSTRACT

The utilization of hash functions and Keyed-Hash Message Authentication Codes (HMAC) are of utmost importance to ensure data integrity and data origin authentication in digital communications. Until recently, protocols used in the Internet, such as the Internet Key Exchange (IKE), Internet Protocol Security (IPSec) and Transport Layer Security (TLS), employed HMAC in conjunction with MD5 and SHA-1 hash functions. However, the finding of efficient collision search attacks against those hash algorithms has demanded the utilization of more secure standards, such as SHA-2. In face of this current trend, an FPGA-based high-performance HMAC/SHA-2 processor is presented in this paper. When computing a message digest and a MAC for 1-block message, the proposed processor can reach throughput higher than 760Mbps and 1.5Gbps, respectively. Regardless of the operation performed, the energy consumption is always below $5\mu J$ per message block. These results are fundamental to provide higher security levels for both mobile devices and servers, which require high-speed and low-energy implementations.

Keywords

Application Specific Processor, Cryptography, SHA-2, Hash Function, HMAC, Keyed-Hash Message Authentication Code

1. INTRODUCTION

The growing demand of secure high-speed digital communications has demanded the inclusion of cryptographic primitives into system design. Since cryptographic processing is usually not lightweight, some applications delegate those tasks to specific hardware modules. One of the most important requirements to be satisfied in secure communications is data integrity and data origin authentication. In order to satisfy those security requirements, the U.S. National Institute of Standards and Technology (NIST) recommends the use of the Keyed-Hash Message Authentication Code (HMAC) [20] based on the Secure Hash Standard (SHS) [22].

Several standards and protocols employ SHS and HMAC. For instance, the Digital Signature Standard (DSS) [21] utilizes SHA to compute message digests and generate random numbers. Furthermore, HMAC along with SHS is employed in the Transport Layer Security Protocol (TLS) [8] and Security Architecture for the Internet Protocol (IPsec) [14]. More specifically, those algorithms are used to perform data origin authentication and integrity verification for the IPSec Authentication Header (AH) [12], Encapsulating Security Payload (ESP) [13], and Internet Key Exchange Protocol (IKE and IKEv2) [10].

The security of the aforementioned algorithms and protocols depends directly on the cryptographic strength of the underlying hash functions. Since efficient collision search attacks [27, 26] against SHA-1 [22] and MD5 [23] hash functions have been found, there has been a trend to move to more secure hash algorithms of SHS, such as SHA-2 [22]. Even though there are specifications [11, 8] to use HMAC together with the SHA-2 family of hash functions, hardware implementations of this more secure conjunction of algorithms have been lagging behind. As far as we are concerned, all hardware designs of HMAC found in the literature [17, 24, 25, 18, 9, 16, 15] are based on the already broken SHA-1 and MD5 algorithms.

In order to address the ongoing demand of higher levels of security, a high-performance HMAC/SHA-2 processor is introduced in this paper. More precisely, four hardware modules are implemented on an Altera Stratix III FPGA [4] comprising the entire SHA-2 family of hash functions: HMAC/SHA-224, HMAC/SHA-256, HMAC/SHA-384, and HMAC/SHA-512. Each of them is fully compliant with NIST specifications [22, 20], and capable of computing both message digests and message authentication codes (MACs). Besides, mobile devices such as cell-phones and PDAs may require small and low power implementations. In contrast, servers may prioritize high throughput in detriment of energy consumption. On top of that, the utilization of more secure standards may increase both implementation requirements and power/energy consumption, as well as impact negatively the system performance. Therefore, in order to better characterize the proposed HMAC/SHA-2 modules, a thorough analysis is performed taking into account implementation area, memory and register requirements, frequency of operation, throughput, and energy consumption.

The remainder of this paper is organized as follows. Section 2 lists related work, whereas Section 3 describes the SHA-2 and HMAC algorithms. Section 4 introduces the architectures of the hardware modules and experimental results are shown in Section 5. In Section 6 the proposed hardware modules are compared with third-party implementations. Our conclusions are presented in Section 7.

2. RELATED WORK

One of the earliest hardware designs of HMAC based on the SHA-1 hash function is reported in [24]. In fact, the module is capable of performing both HMAC and SHA-1, but low throughput is achieved. A single-chip processor for the IPsec protocol is introduced in [17]. This processor implements both HMAC/SHA-1 and AES [19] cores on a Xilinx Virtex-E FPGA [1] therefore providing full hardware support for IPsec. Although AES is currently used as a standard, the use of SHA-1 may not longer satisfy the security requirements of several applications. The authors in [25] propose an HMAC processor which integrates both SHA-1 and MD5 algorithms, and is implemented on an Altera Apex 20K [2]. Given the existence of efficient collision search attacks against SHA-1 and MD5, there is a crescent demand for more secure hash functions. In [18], a high-performance HMAC/SHA-1 design is presented and implemented on a Xilinx Virtex-E FPGA. It relies on pipelining techniques to achieve throughput in the order of Gbps. However, the main drawback of this design is its large implementation area. In contrast, an area-efficient HMAC hardware implementation is introduced in [9] which achieves high throughput. This performance gain is due to a technique denoted as rolling loop, which fuses algorithms loops together. Unfortunately the authors do not provide many details on the design architecture, so that it is difficult to figure out the implemented functionalities and whether it is fully compatible with the HMAC specification [20, 22]. An Application Specific Integrated Circuit (ASIC) implementation of HMAC, also based on SHA-1, is presented in [16]. Even though this design is implemented as an ASIC, it achieves low throughput. Finally, yet another HMAC design is proposed in [15] and implemented on a Virtex-II [3] device. Given that this design comprises three hash functions (SHA-1, MD5, and RIPEMD), it occupies a very large implementation area. Further comparisons of our proposed design with related work is presented in Section 6.

Although the aforementioned researchers have explored architectural improvements to increase throughput and reduce implementation area, they have mainly focused on hash algorithms which were secure at the time of invention but have now been broken. Currently, HMAC based on SHA-1 and MD5 no longer satisfy the security requirements of many applications. Hence, the focus of this paper is to provide an HMAC processor based on the SHS, and comprising the entire SHA-2 family of hash functions. As a consequence, the proposed processor provides applications with much higher levels of security than previous work. Additionally, another goal of this work is to evaluate the impact on area, performance and energy consumption resulting from the choice to use more secure standards such as SHA-2 with HMAC.

3. ALGORITHM DESCRIPTION

One of the goals of the HMAC algorithm is to be independent of a given hash function, so that the latter can be easily replaced with faster and more secure algorithms. Besides, the underlying hash function constitutes the core of the HMAC algorithm, and dictates its security level. Since this work proposes an HMAC processor based on SHA-2, both algorithms are presented in this section. Further details for HMAC [20] and SHA-2 [22] can be found in their original specifications

3.1 SHA-2 Algorithm

The SHA-2 family of hash functions [22] comprises four algorithms, namely, SHA-224, SHA-256, SHA-384, and SHA-512. Since SHA-224 and SHA-256 have several commonalities, the shortcut SHA-224/256 refers for both hash functions. Likewise, SHA-384/512 refers to both SHA-384 and SHA-512. SHA-224/256 can process messages up to 2^{64} bits long, whereas SHA-384/512 can process up to 2^{128} bits. The algorithm output is called message digest, which has length L. L varies according to the algorithm used. For instance, $L = 224$ for SHA-224, $L = 256$ for SHA-256, $L = 384$ for SHA-384, and $L = 512$ for SHA-512. Furthermore, the datapath width (in bits) of these algorithms are denoted by D. Precisely, $D = 32$ for SHA-224/256 and $D = 64$ for SHA-384/512. The execution of SHA-2 algorithms is divided into two parts: Pre-processing and Hash computation. A list of SHA-2 parameters is presented in Figure 1, whereas SHA-2 symbols are listed below:

B : SHA-2 input block size (in bits);
L : SHA-2 message digest size (in bits);
D : Datapath width (in bits);
N : Number of message blocks;
i : Message block index, where $1 \leq i \leq N$;
j : Number of algorithm iterations;
t : Iteration index, where $0 \leq t \leq j-1$;
$M^{(1)}, ..., M^{(N)}$: Message blocks;
$H_0^{(0)}, ..., H_7^{(0)}$: Initial hash values;
$H_0^{(i)}, ..., H_7^{(i)}$: Intermediate hash values;
$W_0, ..., W_j$: Message schedule words;
$a, ..., h$: Working variables;
$K_0, ..., K_j$: Constants.

Table 1: SHA-2 algorithm parameters

Parameter	SHA-			
	224	256	384	512
B	512		1024	
L	224	256	384	512
D	32		64	
j	64		80	

3.1.1 Pre-processing

The SHA-2 input block size depends on the algorithm used. The input block size is denoted as B, and is equal to 512 bits for SHA-224/256 and to 1024 bits for SHA-384/512. The pre-processing stage first splits the original message into N blocks, namely $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Each block has B bits. Then, if the message length is not a multiple of the underlying block size, message padding must be performed. Next, eight initial hash values, $H_0^{(0)}, ..., H_7^{(0)}$, are set as listed in the specifications [22]. Each algorithm uses a distinct set of initial hash values.

3.1.2 Hash Computation

The hash computation is based on operations over D-bit words. The number of iterations performed by the algorithm is given by j. For SHA-224/256, $j = 64$, whereas for SHA-384/512, $j = 80$. Actually, j can be considered to represent the number of D-bit words processed by the algorithm. More specifically, the SHA-2 algorithms comprise j message schedule words ($W_0, ..., W_j$), eight working vari-

ables (a, b, c, d, e, f, g, h), and eight intermediate hash values $(H_0^{(i)}, ..., H_7^{(i)})$. As specified in [22], SHA-224/256 and SHA-384/512 utilize j constants $(K_0, ..., K_j)$, where each of them is D bits wide. Additionally, six logical functions are employed, as shown below. $ROR^n(x)$ and $SHR^n(x)$ correspond to, respectively, a rotation and a shift of x by n bits to the right. Besides, $\oplus$ represents the bitwise XOR operation, $\wedge$ the bitwise AND operation, and $\bar{x}$ the bitwise complement of x.

SHA-224/256:

$$\begin{aligned} Ch(x,y,z) &= (x \wedge y) \oplus (\bar{x} \wedge y), \\ Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z), \\ \sum_0(x) &= ROR^2(x) \oplus ROR^{13}(x) \oplus ROR^{22}(x), \\ \sum_1(x) &= ROR^6(x) \oplus ROR^{11}(x) \oplus ROR^{25}(x), \\ \sigma_0(x) &= ROR^7(x) \oplus ROR^{18}(x) \oplus SHR^3(x), \\ \sigma_1(x) &= ROR^{17}(x) \oplus ROR^{19}(x) \oplus SHR^{10}(x). \end{aligned}$$

SHA-384/512:

$$\begin{aligned} Ch(x,y,z) &= (x \wedge y) \oplus (\bar{x} \wedge y), \\ Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z), \\ \sum_0(x) &= ROR^{28}(x) \oplus ROR^{34}(x) \oplus ROR^{39}(x), \\ \sum_1(x) &= ROR^{14}(x) \oplus ROR^{18}(x) \oplus ROR^{41}(x), \\ \sigma_0(x) &= ROR^1(x) \oplus ROR^8(x) \oplus SHR^7(x), \\ \sigma_1(x) &= ROR^{19}(x) \oplus ROR^{61}(x) \oplus SHR^6(x). \end{aligned}$$

For each message block i, $1 \leq i \leq N$, a four-step digest round is performed as follows:

Step 1: Initialize the eight working variables

$$\begin{aligned} a = H_0^{(i-1)}, \quad b = H_1^{(i-1)}, \quad c = H_2^{(i-1)}, \quad d = H_3^{(i-1)}, \\ e = H_4^{(i-1)}, \quad f = H_5^{(i-1)}, \quad g = H_6^{(i-1)}, \quad h = H_7^{(i-1)}. \end{aligned}$$

Step 2: Prepare the message schedule

$$W_t = \begin{cases} M_t^{(i)}, & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}, & 16 \leq t \leq j-1. \end{cases}$$

Step 3: For $t = 0$ to $j - 1$ do:

$$\begin{aligned} &T_1 = h + \textstyle\sum_1(e) + Ch(e,f,g) + K_t + W_t, \\ &T_2 = \textstyle\sum_0(a) + Maj(a,b,c), \\ &h = g, \\ &g = f, \\ &f = e, \\ &e = d + T_1, \\ &d = c, \\ &c = b, \\ &b = a, \\ &a = T_1 + T_2, \end{aligned}$$

Step 4: Compute the i^{th} intermediate hash value $H^{(i)}$:

$$\begin{aligned} H_0^{(i)} = a + H_0^{(i-1)}, \quad H_1^{(i)} = b + H_1^{(i-1)}, \\ H_2^{(i)} = c + H_2^{(i-1)}, \quad H_3^{(i)} = d + H_3^{(i-1)}, \\ H_4^{(i)} = e + H_4^{(i-1)}, \quad H_5^{(i)} = f + H_5^{(i-1)}, \\ H_6^{(i)} = g + H_6^{(i-1)}, \quad H_7^{(i)} = h + H_7^{(i-1)}. \end{aligned}$$

After all N blocks of message M are processed, the final message digest is obtained by concatenating the hash values $(H_0^{(i)}, ..., H_7^{(i)})$. More precisely, the message digest for each algorithm is given by the concatenations shown below. The concatenation of words is represented by the symbol $||$.

SHA-224:

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}.$$

SHA-256 and SHA-512:

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}||H_7^{(N)}.$$

SHA-384:

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}.$$

3.2 HMAC Algorithm

The HMAC algorithm processes two inputs, a cryptographic key and a message, to produce message authentication code (MAC). The combination of HMAC with SHA-2 is denoted as HMAC/SHA-2. Individual combinations of HMAC with the four SHA-2 algorithms are denoted as HMAC/SHA-224, HMAC/SHA-256, HMAC/SHA-384, and HMAC/SHA-512. Due to certain commonalities between SHA-224 and SHA-256, as well as between SHA-384 and SHA-512, the algorithms can also be referred to as HMAC/SHA-224/256 and HMAC/SHA-384/512. HMAC symbols are listed below:

$$\begin{aligned} B &: \text{SHA-2 input block size (in bits);} \\ L &: \text{SHA-2 message digest size (in bits);} \\ Key &: \text{Secret key of the communicating parties;} \\ K &: \text{Size of the } Key \text{ (in bits);} \\ K_0 &: Key \text{ after any necessary pre-processing;} \\ Ipad &: \text{(Inner pad) Byte } \texttt{0x36} \text{ repeated } B/8 \text{ times;} \\ Opad &: \text{(Outer pad) Byte } \texttt{0x5C} \text{ repeated } B/8 \text{ times;} \\ Text &: \text{The data on which the HMAC is calculated;} \\ Hash(V) &: \text{Hash of variable/value } V; \\ ||(0..)_z &: \text{Padding with } z \text{ zeros.} \end{aligned}$$

Parameters B and L are inherited from SHA-2. The *Text* can be n bits long, where $0 \leq n < 2^{B/8} - B$, whereas the size of a MAC is L bits long. The HMAC algorithm is consisted of seven steps as shown below. Notice that the hashes computed in *Step IV* and *Steps VII* can be split into two parts to facilitate its implementation.

Step I: Pre-process Key as follows:

$$K_0 = \begin{cases} Key, & K = B, \\ Key||(0..)_{B-K}, & K < B, \\ Hash(Key)||(0..)_{B-L}, & K > B. \end{cases}$$

Step II: Compute $(K_0 \oplus Ipad)$.

Step III: Do $(K_0 \oplus Ipad)||Text$.

Step IV: $Hash((K_0 \oplus Ipad)||Text)$.

Step V: Compute $(K_0 \oplus Opad)$.

Step VI: Do $(K_0 \oplus Opad)||Hash((K_0 \oplus Ipad)||Text)$.

Step VII: $Hash((K_0 \oplus Opad)||Hash((K_0 \oplus Ipad)||Text))$.

Step VII produces the MAC of the message digest. A common practice is to truncate the MAC by using only its t leftmost bits. In this work, we consider that the MAC is L bits long, and any truncation is performed at the user level.

4. HARDWARE DESIGN

Individual HMAC hardware modules were devised for each of the four SHA-2 algorithms. The idea behind was to have precise results on implementation area, memory and register requirements, throughput, and energy consumption of each hardware module. The following sections presents the hardware designs of the SHA-2 and HMAC cores.

4.1 SHA-2 Core

The hardware design of the SHA-2 algorithm consists of shift-registers, logical operations, D-bit adders, and a memory to store the algorithm constants. The register requirements are 1024 bits for SHA-224/256, and 2048 bits for SHA-384/512. SHA-224/256 and SHA-384/512 constants (K_t) are stored in block RAMs, and require, respectively, 2048 and 5120 bits. As shown in Figure 1, the SHA-2 architecture can be divided into four main blocks: Intermediate Hash Computation, Compressor, Message Scheduler, and Constants Memory.

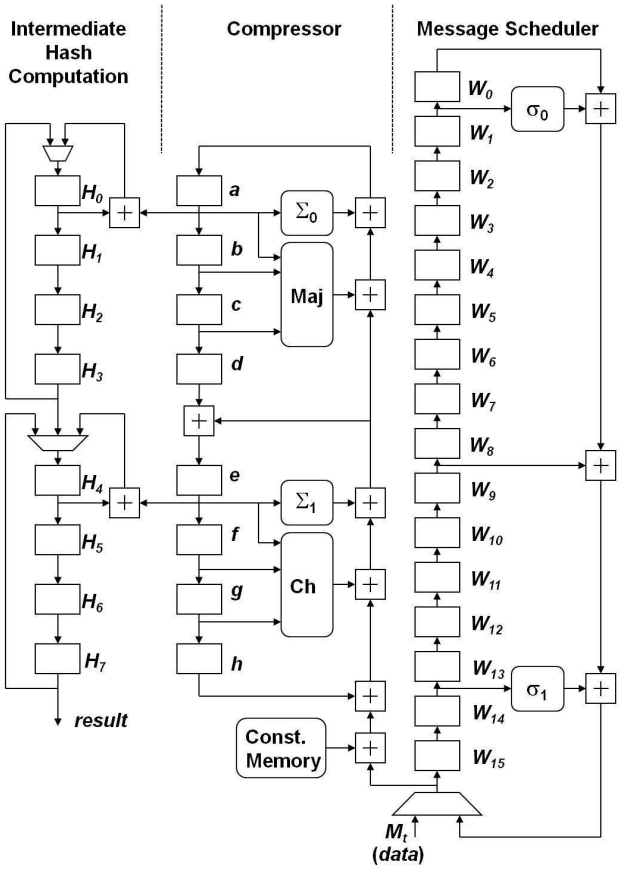

Figure 1: SHA-2 architecture

The message scheduler's registers $W_0, ..., W_{15}$ can be initialized either in serial or in parallel. Serial initialization requires the first 16 words M_t of the message M to be shifted into the module; this processing takes 16 clock cycles. It is utilized when writing a new message or key to the SHA-2 core through the HMAC module. For instance, it can be a message being hashed, a long key being pre-processed, or a text being processed as part of a MAC computation. Parallel initialization is used by the HMAC module to load internally stored values directly into registers $W_0, ..., W_{15}$. Simultaneously, the constants memory provides the initialization values for the working variables $(a, ..., h)$. Initial hashes $(H_0, ..., H_7)$ are also set within this period of time. The initialization of working variables and initial hashes takes one clock cycle. When initialization is complete, the compressor employs registers $a, ..., h$, as well as W_t and K_t to determine the new values of $a, ..., h$. As described in *Step 3* of SHA-2, the algorithm takes t iterations and is controlled internally by an iteration counter. Precisely, SHA-224/256 and SHA-384/512 utilize, respectively, utilizes 64 and 80 iterations. In each of these iterations, registers $W_0, ..., W_{15}$ and $a, ..., h$ are shifted in the direction of the arrows shown in Figure 1.

After t iterations, the intermediate hash computation is performed. It would be possible to execute this operation in a single clock cycle, but that would demand eight adders operating in parallel. In order to save implementation area, only two adders are utilized. As a consequence, the computation of the intermediate hash is spread over the last 4 iterations by computing two additions per clock cycle. More precisely, the additions are performed when $t = 60, ..., 63$ for SHA-224/256. For instance, in SHA-224/256, H_3 and H_7 are computed when $t = 60$, H_2 and H_6 are computed when $t = 61$, and so on. The same strategy is followed by SHA-384/512, but the additions are executed when $t = 76, ..., 79$.

In the case of hashing a multi-block message, a new execution cycle initiates. Then, if serial load is used, 16 more words M_t must be shifted into the module. If parallel load is used, the HMAC module reloads registers $W_0, ..., W_{15}$ with the appropriate value. After that, the same procedure described above is re-executed. When the hash is complete, the message digest is available in registers $H_0, ..., H_7$. Again, the message digest may correspond to either a single hash computation or to any of the hashes computed in the HMAC algorithm. In the case of a single hash computation, the result is read serially. Since a D-bit word is output per read operation, L/D iterations are necessary. Specifically, SHA-224, SHA-256, SHA-384 and SHA-512 require 7, 8, 6, and 8 read operations respectively. In the case of the HMAC processing, registers $H_0, ..., H_7$ are read in parallel by the HMAC core.

4.2 HMAC Core

The HMAC hardware module was designed to cope with both message digest and MAC computations, to process long messages and keys of different sizes ($K \leq B$ and $K > B$), as well as to perform pre-processing of long keys as necessary. Moreover, the module is capable of reusing keys therefore increasing its throughput. As illustrated in Figure 2, the proposed HMAC architecture consists of a SHA-2 core, multiplexors, logical operations, and registers. Precisely, three registers are employed, namely $K_0\_Ipad\_Hash$, $K_0\_Ipad\_Text\_Hash$, and $K_0\_Opad\_Hash$. Actually, key reuse is supported the utilizing registers $K_0\_Ipad\_Hash$ and $K_0\_Opad\_Hash$ in further computations. Moreover, register $K_0\_Opad\_Hash$ also works as a temporary key storage during the HMAC processing. The size of those registers vary according to the hash algorithm used, as listed in Table 2. In Figure 2, the SHA-2 registers $W_0, ..., W_{15}$, $a, ..., h$, and $H_0, ..., H_7$ are referred to as W, $a..h$, and H, respectively.

Table 2: Register requirements of HMAC

Register	HMAC/SHA- 224 (bits)	256 (bits)	384 (bits)	512 (bits)
K_0_$Ipad$_$Hash$	256		512	
K_0_$Opad$_$Hash$	512		1024	
K_0_$Ipad$_$Text$_$Hash$	224	256	384	512

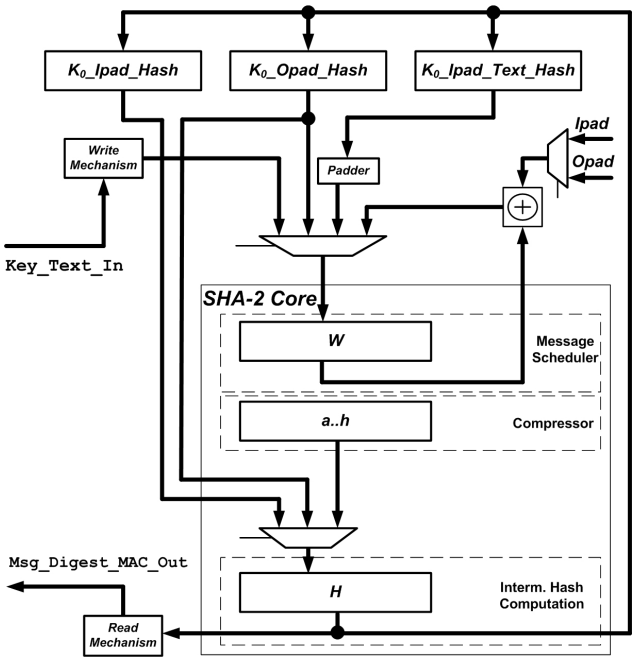

Figure 2: HMAC architecture

In the case of hash computation, the HMAC module allows the user to interface directly with the SHA-2 core through the ports `Key_Text_In` and `MsgDigest_MAC_Out`. As a result, the interfacing with the module is performed exactly as described in Section 4.1. The HMAC processing is divided into five stages, namely `NewKeyHash`, `KeyIpadHash`, `TextHash`, `KeyOpadHash`, and `MACHash`. The stages are executed in that order, but not all stages are necessarily used. The size of the key, the size of the text, and whether a key is being reused will determine the number of stages needed. In any case, whenever writing a new key or the message text to the HMAC module, or reading a MAC result, the same interfacing procedures described in Section 4.1 are followed. It is important to mention that HMAC module interface is similar to the Avalon interface specification [5]. As a consequence, it can be easily implemented as a peripheral of a Nios II processor [6] with minor (or even no) modifications.

If a new key is used and $K = B$ no padding is needed, and $K_0 = Key$. However, if $K < B$, padding is performed to create K_0 as described in *Step I*. Since the key is written into W, this register is temporarily used to perform any necessary pre-processing. The execution proceeds with the `KeyIpadHash` stage. Note that, since `KeyOpadHash` also needs K_0, its value is temporarily stored in K_0_$Opad$_$Hash$; otherwise it would be erased by the hash computation. Stage `KeyIpadHash` performs one hash computation, which corresponds to the first part of *Step IV*. The result is stored in register K_0_$Ipad$_$Hash$. Next, the message text is written to the module, followed by the execution of stage `TextHash`. This stage performs the second part of *Step IV*, and may be re-executed as long as there are message blocks to be processed. More specifically, if the message is N blocks long, `TextHash` is executed N times. Its result is stored in K_0_$Ipad$_$Text$_$Hash$. In the sequence, `KeyOpadHash` is executed, which corresponds to the first part of *Step VII*. Before of its execution, however, it loads K_0 previously stored in K_0_$Opad$_$Hash$ onto W. This stage computes a single hash and store its result back in K_0_$Opad$_$Hash$. Notice that at this point, both K_0_$Ipad$_$Hash$ and K_0_$Opad$_$Hash$ have the processed K_0, which can be employed in any future key reuse. Finally, stage `MACHash` performs the second part of *Step 7*. In order to accomplish that, it loads K_0_$Ipad$_$Text$_$Hash$ onto W and computes the last hash of the HMAC algorithm. Upon the completion of this stage, the MAC is available in H. Moreover, it can be read identically to a message digest, as described in Section 4.1.

In the case of new long key ($K > B$), the processing starts in stage `NewKeyHash` to perform the key pre-processing (*Step I*). Before each `NewKeyHash` execution, one block of the key must be written to W. Furthermore, if a k-block key is used, this stage is executed k times. After `NewKeyHash` completion, K_0 becomes available in register H. At this point, K_0 is padded with zeros and loaded onto register W, thus allowing for stage `KeyIpadHash` to start. The remaining HMAC processing is identical to the one describe above.

Finally, when a key is reused, the module take advantage of previous computations to speed up the HMAC execution. The only computation needed is the one dependent on the new message being processed. Since K_0_$Ipad$_$Hash$ and K_0_$Opad$_$Hash$ are already stored within the module, the first part of *Steps IV* and *VII* do not need to be computed. Hence, the processing starts in the `TextHash` stage to perform the second part of *Step IV*. Again, this stage is executed as long as there are message blocks to be processed. In other words, if the message is N blocks long, `TextHash` is executed N times. In this case, the result of `TextHash` is not stored in K_0_$Ipad$_$Text$_$Hash$, but loaded onto W instead. Next, K_0_$Opad$_$Hash$ is loaded onto H, and stage `MACHash` performs the second part of *Step 7*. After completing the execution of `MACHash`, the MAC is available in H.

5. EXPERIMENTAL RESULTS

In order to obtain the experimental results, the proposed HMAC processor has been described in VHDL and implemented on an Altera Stratix III EP3SE50F484C2 FPGA. The tool utilized in the description, synthesis, simulation and energy estimation was QuartusII version 9.0 [7].

5.1 Area and Frequency Analysis

Implementation area is measured as the number of Adaptive Look-Up Tables (ALUTs) employed to implement the processor on the FPGA. Frequency of operation is reported by QuartusII after performing place-and-route for the target FPGA, and is measured in MHz. Table 3 lists the area requirements and frequency of operation of the HMAC/SHA-2 processor.

Table 3: Area and Frequency Results

Module	Area (ALUTs)	Frequency (MHz)
HMAC/SHA-224	2262	138.97
HMAC/SHA-256	2347	138.10
HMAC/SHA-384	4240	120.83
HMAC/SHA-512	4601	116.04

The modules HMAC/SHA-224 and HMAC/SHA-256 occupies 2262 and 2347 ALUTs, respectively. Recall that SHA-224 and SHA-256 differ in two aspects: 1) initial hash values, and 2) message digest size. Since the initial hash values do not differ in size and number, the difference in the area requirements is explained by the message digest size: 224 and 256 bits, respectively, for SHA-224 and SHA-256. As a consequence, HMAC/SHA-224 needs simpler logic to read shorter MACs, and that justifies its smaller implementation area. Similar behavior is observed for HMAC/SHA-384 and HMAC/SHA-512. HMAC/SHA-384 occupies 4240 ALUTs, in contrast to the 4601 ALUTs employed by HMAC/SHA-512.

Although the HMAC/SHA-384/512 have a datapath twice as wide as the HMAC/SHA-224/256 ones, the former two modules demand less than twice as much area than the latter ones. Precisely, HMAC/SHA-384 employs 1.88 times as much area as HMAC/SHA-224, whereas HMAC/SHA-512 is 1.96 times as large as HMAC/SHA-256.

The highest frequency is achieved by HMAC/SHA-224, which is capable of performing at 138.97MHz. HMAC/SHA-512 is the more complex module in terms of logic which causes it to have the lowest frequency of operation (116.04MHz). Despite the datapath of HMAC/SHA-384/512 are twice as wide as the ones in HMAC/SHA-224/256, the frequency of operation is not impacted in same ratio. HMAC/SHA-512 module is 16% slower compared to HMAC/SHA-256, whereas HMAC/SHA-384 is 13% slower than HMAC/SHA-224.

5.2 Register and Memory Requirements

The register and memory requirements are presented in Table 4 and were determined by QuartusII after performing place-and-route. As described in Section 4, the nominal register requirements of HMAC/SHA-224, HMAC/SHA-256, HMAC/SHA-384, and HMAC/SHA-512 are 2016, 2048, 3968 and 4096 bits, respectively. The hardware implementation, however, requires additional bits to implement an iteration counter and some other registers needed in the algorithm processing. Precisely, to determine if it is computing a hash or an HMAC, if it is processing key or text, if it is the first/last block of a message/key, if it is a new key, and if it is a long key, and which HMAC stage is being performed. As a result, HMAC/SHA-224/256 and HMAC/SHA-384/512 utilizes, respectively, 19 and 20 more bits than the nominal register requirements.

The memory requirements after place-and-route is exactly the same as specified in Section 4. HMAC/SHA-224/256 demands 2048 bits, whereas HMAC/SHA-384/512 demands 5120 bits.

Table 4: Register and Memory Requirements

Module	Register (bits)	Memory (bits)
HMAC/SHA-224	2035	2048
HMAC/SHA-256	2067	2048
HMAC/SHA-384	3988	5120
HMAC/SHA-512	4116	5120

5.3 Throughput Analysis

The throughput of SHA-2 is defined as the amount of text processed per amount of time, and is given in bits per second (bps). It can be determined from the message block size (B), module's frequency of operation (F), and the number of clock cycles (c) needed to compute a message digest. Specifically, the throughput of the hash function is computed as:

$$Throughput_{Hash} = \frac{B * F}{c}. \quad (1)$$

Given that HMAC processor is based on SHA-2, its throughput is a function of the hash throughput. As traditionally considered in the literature, the key is not taken as an input while computing throughput. In other words, although accounting the time necessary to process the key, only the message is considered as input for the HMAC execution.

As described in Section 4, if a new key is used, 3 hashes (`KeyIpadHash`, `KeyOpadHash`, and `MACHash`) are always computed. Besides, if a new k-block key is used ($K > B$), `NewKeyHash` is performed k times to pre-process the key. If ($K \leq B$), `NewKeyHash` is not performed, i.e. $k = 0$. Also, it the message is m blocks long, `TextHash` is executed m times. In sum, when a new key is used, the computation of $3 + m + k$ hash functions are needed. As a consequence, the HMAC throughput using a new key is given by

$$Throughput_{HMAC_{NewKey}} = \frac{m}{3 + m + k} Throughput_{Hash}. \quad (2)$$

If the key is reused, `TextHash` is performed m times and `MACHash` once. As a result, the HMAC throughput is determined by

$$Throughput_{HMAC_{KeyReuse}} = \frac{m}{1 + m} Throughput_{Hash}. \quad (3)$$

The processor throughput, as listed in Table 5, considers the computation of one message digest/MAC using 1-block message (B bits) at the maximum frequency of operation. The execution of SHA-224/256 consumes 65 clock cycles ($c = 65$) to process a 512-bit block ($B = 512$), whereas SHA-384/512 takes 81 clock cycles ($c = 81$) to process a 1024-bit block ($B = 1024$).

As shown in Table 5, independently of the SHA-2 algorithm used, the processor reaches more than 1Gbps to compute one message digest. The highest throughput is due to SHA-384, which is able to deliver 1527.53Mbps. In contrast, the lowest throughput (1087.80Mbps) is achieved when using SHA-256. SHA-224 and SHA-512, in turn, achieve 1094.66 and 1466.97Mbps, respectively.

Table 5: Throughput Results

Module	SHA-2	HMAC, 1-Block Message		
	1-Block Message (Mbps)	1-Block New Key (Mbps)	2-Block New Key (Mbps)	Key Reuse (Mbps)
HMAC/SHA-224	1094.66	273.66	182.44	547.33
HMAC/SHA-256	1087.80	271.95	181.30	543.90
HMAC/SHA-384	1527.53	381.88	254.59	763.76
HMAC/SHA-512	1466.97	366.74	244.50	733.49

Table 5 also lists the HMAC throughput when processing a 1-block message (B bits) in three main cases: 1) When a 1-block new key ($K = B$) is employed; 2) When a 2-block new key ($K = 2B$) is utilized; and 3) When a key is reused. When computing HMAC with a 1-block new key, no key pre-processing is required. Therefore, Equation 2 is used with $m = 1$ and $k = 0$. In this case, HMAC/SHA-256 has the lowest throughput (271.Mbps), whereas HMAC/SHA-384 achieves the highest one (381.88Mbps). HMAC/SHA-224 and HMAC/SHA-512 throughput are, respectively, 273.66 and 366.74Mbps, respectively. If a 2-block key is used, the processor has to pre-process the key. Consequently, `NewKeyHash` needs to be performed twice. Then, Equation 2 is utilized with $m = 1$ and $k = 2$. As a result, the processor throughput is negatively impacted. For example, the throughput of HMAC/SHA-384 and HMAC/SHA-256 become to 254.59 and 181.30Mbps, respectively.

When a key is reused, the throughput is determined by Equation 3 along with $m = 1$. In other words, only `TextHash` and `MACHash` are computed, which increases the throughput. This case leads to a throughput as high as 763.76Mbps, as achieved by HMAC/SHA-384. The lowest throughput in this case is due to HMAC/SHA-256 (543.90Mbps). Notice that reusing the key causes the throughput to be twice as high as the one achieved when using a 1-block new key. Furthermore, since only two hashes are computed, the throughput of HMAC with key reuse is exactly the half of the ones obtained with the hash function.

By examining the throughput of the processor using SHA-224 and SHA-384, it is possible to notice that, in general, SHA-384 is about 40% faster than SHA-224. Also, SHA-512 is about 35% faster than SHA-256. This proposition is also true when computing HMAC. SHA-384/512 have message blocks (1024 bits) twice as large as the ones in SHA-224/256 (512 bits). Moreover, SHA-384/512 modules, which take 81 clock cycles/per block does not take twice as many clock cycles as SHA-224/256 modules, which take 65 clock cycles/per block to compute a hash. As a consequence, one could have the false impression that SHA-384/512 would, at least, duplicate the throughput of SHA-224/256. It is counterintuitive, though, to think that if all implementations run at the same clock frequency, SHA-384/512 would be 61% faster than SHA-224/256. In order to show that, consider Equation 1. Let $F_{(224/256)}$ and $F_{(384/512)}$ be, respectively, the frequency of operation of SHA-384/512 and SHA-224/256. Hence,

$$\frac{Throughput_{(384/512)}}{Throughput_{(224/256)}} = \frac{\frac{1024 F_{(384/512)}}{81}}{\frac{512 F_{(224/256)}}{65}} \approx 1.61 \frac{F_{(384/512)}}{F_{(224/256)}}.$$

Similar analysis can be done for the performance of HMAC. Since HMAC/SHA-224 implementation has a frequency of operation 15% higher than the one obtained in HMAC/SHA-384, the latter is only 40% faster than the former; not 61%. Similarly, the frequency of operation of HMAC/SHA-256 implementation is 19% higher than HMAC/SHA-384, which makes the latter only 35% faster than the former.

5.4 Energy Analysis

Energy consumption is a very important parameter to be considered, mainly in constrained devices. Specifically, energy (E) is determined by the module's power consumption (P) and the amount time spent to compute a given operation (t), i.e. $E = P * t$. In this work, power is measured in Watt (W), energy in Joule (J), and time in second (s).

The power consumption of each module is obtained from the PowerPlay Power Analyzer Tool [7]. The power consumption of the proposed processor is obtained by executing operations using random text and random keys, at its maximum frequency of operation. Then, a signal activity file is created and fed to the power analyzer. In order to reduce the power consumption even further, a two step process was performed. After completing the first synthesis, place-and-route, and simulation, a signal activity file was generated. Next, this file was fed back to the synthesis tool and a second synthesis and place-and-route performed. After a second simulation of the new design, the updated signal activity file was examined by the power analyzer to determine its new power consumption.

The power analyzer tool estimates the FPGA static power, dynamic power, and input/output power. However, since these designs may be implemented in a system-on-a-chip, the input/output power is of minor importance and can be discarded in our analysis. As a consequence, static and dynamic power are used to compute the energy consumption, i.e. $E = (P_{static} + P_{dynamic}) * t$.

Table 6 shows the energy consumption (in μJ) for each module while performing different operations. It includes results for SHA-2 and HMAC processing a 1-block message, and the utilization of a 1-block key, 2-block key, and reusing a key. First considering the computation of hash functions, one can observe that HMAC/SHA-384 and HMAC/SHA-512 have similar energy consumption, respectively, 0.85 and $0.88 \mu J$. The energy consumption of HMAC/SHA-224 and HMAC/SHA-256 are even closer, respectively, $0.47 \mu J$ and $0.48 \mu J$.

If a 1-block new key is used for an HMAC operation, the least energy consumption is achieved with HMAC/SHA-224

Table 6: Energy Results

Module	SHA-2	HMAC, 1-Block Message		
	1-Block Message (μJ)	New 1-Block Key (μJ)	New 2-Block Key (μJ)	Key Reuse (μJ)
HMAC/SHA-224	0.47	1.64	2.43	0.83
HMAC/SHA-256	0.48	1.67	2.48	0.85
HMAC/SHA-384	0.85	3.09	4.59	1.55
HMAC/SHA-512	0.88	3.18	4.73	1.60

($1.64\mu J$). In the case of using HMAC/SHA-512, $3.18\mu J$ are consumed by the module. When a 2-block new key is used, the energy consumption for HMAC/SHA-224 and HMAC/SHA-512 increase to 2.43 and $4.73\mu J$, respectively. This represents an average increase of 49% in the energy demand. Finally, when a key is reused, the energy consumption of HMAC/SHA-224 and HMAC/SHA-512 decreased, respectively, to 0.83 and $1.60\mu J$. Reusing a key reduces the energy demands in about 50% compared to the use of a 1-block new key. Compared to a 2-block new key, there is a 66% reduction in the energy consumption.

Furthermore, HMAC/SHA-384 utilizes 81% more energy than HMAC/SHA-224 when computing a message digest. In turn, HMAC/SHA-512 consumes 87% more energy compared to HMAC/SHA-256. When a MAC computation is performed, HMAC/SHA-384 utilizes in the average 88% more energy than HMAC/SHA-224, whereas HMAC/SHA-512 demands about 90% more energy than HMAC/SHA-256.

By performing a cross-comparison between Tables 5 and 6, one may realize that the modules with the highest throughput do not have the highest energy consumption. That happens due to the way throughput is computed, which does not consider module interfacing. However, in order to be able to simulate the module and acquire the power estimates, data must be sent to and results read from the modules. As a result, the energy spent with interfacing is present in the energy consumption of the modules. Specifically, HMAC/SHA-256 requires one extra iteration to read the MAC than HMAC/SHA-224. Thus, although the former has a lower throughput, it ends up having a higher energy consumption than the latter. Similarly, HMAC/SHA-512 needs two extra iterations to read a MAC than HMAC/SHA-384, which demands more energy. This two extra iterations also explains the wider gap between the energy requirements of HMAC/SHA-512 and HMAC/SHA-384, compared to the one between HMAC/SHA-256 and HMAC/SHA-224.

6. COMPARISONS WITH THIRD-PARTY IMPLEMENTATIONS

Besides the analysis based on the Stratix III FPGA implementation, it is interesting to compare the modules with third-party cores available in the literature. Notice, however, that the main goal is not to show that the proposed processor has lower area or higher throughput than third-party implementations. Actually, that would lead to unfair comparisons and misleading results. All previous hardware implementations of HMAC employed simpler and less secure hash functions, such as SHA-1 and MD5. The SHA-2 algorithm, in turn, is more secure and also much more complex than SHA-1 and MD5 ones. More precisely, compared to SHA-1 algorithm, SHA-2 utilizes more initialization hash values and constants, employs more complex functions, and demands a more elaborated compressor and message scheduler. Thus, it is expected for SHA-2 to utilize more area and have longer critical paths than SHA-1 and MD5.

Since this is, to the best of our knowledge, the first hardware implementation of HMAC based on SHA-2, the focus of this cross-comparison is to analyze how employing a more secure hash function impacts implementation area, frequency of operation, and throughput of the HMAC processor. It is also analyzed the efficiency of each module in terms of throughput per unit of implementation area. This metric is referred to as *throughput/area*. It is important to note that some HMAC implementations include more than one hash function and other algorithms. Besides, it is important to keep in mind that some designs adopt architectural techniques, such as rolling loop and pipelining, to improve performance.

Moreover, comparisons become even more challenging when different devices are used. FPGAs have different internal architectures and synthesis tools will report implementation area according to the device's internal elements, e.g. Logical Elements (LEs), Look-up Tables (LUTs), Adaptive Look-up tables (ALUTs) and Slices. In order to make possible a cross-comparison with third-party cores, the proposed HMAC processor was re-synthesized to target the same devices employed by other implementations. Specifically, the FPGAs utilized in the cross-comparison are: Altera Apex 20K [2], Xilinx Virtex-E [1], and Xilinx VirtexII [3]. Table 7 list the implementation results of several related works, including FPGAs from different vendors.

6.1 Altera Apex 20K Implementations

By analyzing implementations utilizing an Altera Apex 20K FPGA, one can observe that the design of Wang et al. [25] occupies 5329 LEs. This design is capable of performing HMAC with both SHA-1 and MD5. Although HMAC/SHA-224 occupies 70% more area (9037 LEs) than [25], its throughput is 4.31 times higher (149.54Mbps) than the one achieved by the latter (34.7Mbps). Furthermore, HMAC/SHA-512 utilizes 312% more area, but is throughput (180.72Mbps) is 5.2 times higher than the one reported in [25]. Moreover, all proposed modules have higher frequencies of operation compared to [25]. If the efficiency in terms of throughput/area is taken into account, the maximum efficiency is achieved by the module HMAC/SHA-224, which delivers 16.55Kbps/LE. In addition, HMAC/SHA-512 achieves 8.23Kbps/LE. Furthermore, the implementation in [25] leads to a low throughput/area (6.51Kbps/LE), which makes it the least efficient HMAC module.

Table 7: Cross-Comparison of HMAC Implementations

Design	Area (α)	Frequency (MHz)	Throughput (Mbps)	Throughput/Area (Kbps/α)	FPGA Vendor & Device
Wang et al. [25] (HMAC/SHA-1/MD5)	5329	21.96	27.7 – 34.7	5.20 – 6.51	Altera Apex 20K, EP20K1000EBC652
Proposed					
HMAC/SHA-224	9037	37.97	149.54	16.55	Altera Apex 20K,
HMAC/SHA-256	9231	35.55	140.01	15.17	EP20K1000EBC652
HMAC/SHA-384	21541	28.86	182.42	8.47	
HMAC/SHA-512	21965	28.59	180.72	8.23	
Michail et al. [18] (HMAC/SHA-1)	6011	62	1587**	264.02	Xilinx Virtex-E, XCV3200EBG1156
Yiakoumis et al. [9] (HMAC/SHA-1/MD5)	686	111	710.4*	1035.57	Xilinx Virtex-E, XCV1600EBG560
Mcloone et al. [17] (HMAC/SHA-1 & AES)	7247	50	62.4 – 78	8.61 – 10.76	Xilinx Virtex-E, XCV1000E
Proposed					
HMAC/SHA-224	3375	46.62	183.62	54.41	Xilinx Virtex-E,
HMAC/SHA-256	3463	48.12	189.52	54.73	XCV1600EBG560
HMAC/SHA-384	6065	41.89	264.79	43.66	
HMAC/SHA-512	6483	41.93	265.04	40.88	
Kahn et al. [15] (HMAC/SHA-1/MD5/RIPEMD160)	14911	43.47	137.40	9.21	Xilinx Virtex-II, XCV2V4000
Proposed					
HMAC/SHA-224	3492	61.58	242.53	69.45	Xilinx Virtex-II,
HMAC/SHA-256	3608	59.66	234.97	65.12	XCV2V4000BF957
HMAC/SHA-384	6247	56.21	355.30	56.88	
HMAC/SHA-512	7017	52.17	329.77	47.00	

α: Area is measured in logical elements (LEs) and Slices, respectively, for Altera and Xilinx devices;
$*$ Rolling loop; $**$ Pipelining

6.2 Xilinx Virtex-E Implementations

Now considering a Xilinx Virtex-E device, Michail et al. [18] introduces a design to perform HMAC with SHA-1. It has the highest throughput for HMAC hardware implementation reported in the literature (1587Mbps) when running at 62MHz. Such a high throughput is due to the employment of two SHA-1 cores and pipelining techniques, which ends up demanding a large implementation area (6011 slices). Yiakoumis et al. [9] proposes a design capable of performing both HMAC/SHA-1 and HMAC-MD5 which occupies 686 slices, and achieves a throughput of 710.4Mbps. This high throughput is due to architectural improvements such as rolling loop. Unfortunately, [9] does not provide a discussion on how such a small area was achieved using rolling loop. Furthermore, the capabilities of the hardware module are not specified, which makes it hard to know whether the module is fully compatible with the HMAC specification [20]. Yet another implementation of HMAC/SHA-1 on Virtex-E is presented by Mcloone et al. [17]. In fact, this design also includes an AES core, which is implemented separately from the HMAC/SHA-1 one. Unfortunately, the area requirements of the individual modules are not reported, so that it is not possible to determine the area requirements of HMAC alone. Both cores can operate independently from each other, which results in a minor impact (if any) on the throughput. Actually, the throughput achieved by [17] is 78Mbps when running at 50MHz. Finally, when the proposed HMAC/SHA-224 processor is implemented on a Virtex-E device, it occupies 3375 slices. This means it is 44% smaller than [18] and 392% bigger than [9]. Also, its throughput is 183.62Mbps and its efficiency in terms of throughput/area is 54.41Kbps/slice. The implementation in [18] has a throughput/area ratio of 10.76Kbps/slice. Besides, [18] and [9] have increased efficiency in terms of throughput/area, respectively, 264.02 and 1035.57Kbps/slice. However, that is result of the use of pipelining, parallelism, and rolling loop techniques.

6.3 Xilinx Virtex-II Implementations

The design proposed by Kahn et al. in [15] utilizes a Virtex-II FPGA. Due to the several hash algorithms included (SHA-1, MD5, and RIPEMD-160), this module utilizes a large implementation area (14911 slices). Besides, this implementation achieves a throughput of 137.40Mbps when operating at 43.47MHz. Thus, its efficiency in terms of throughput/area is 9.21Kbps/slice. The proposed HMAC/SHA-224 module utilizes 3492 slices, which means it is 77% smaller than [15]. HMAC/SHA-224 has a throughput of 242.53Mbps when operating at 69.45MHz, which leads to a throughput/area of 69.45Kbps/slice. HMAC/SHA-512 occupies 7017 slices and achieves a throughput of 329.77Mbps, resulting in a throughput/area ratio of 47Kbps/slice. This result means that the HMAC/SHA-512 module leads to a higher efficiency in terms throughput/area, and is 53% smaller than the implementation in [15].

7. CONCLUSIONS

This work introduces a high-performance HMAC processor based on SHA-2 family of hash functions. Precisely, SHA-224, SHA-256, SHA-384, and SHA-512 are implemented, resulting in four HMAC hardware modules. The implementations are thoroughly analyzed in terms of implementation area, memory and register requirements, frequency of operation, throughput and energy consumption.

To the best of our knowledge, this is the first hardware implementation of HMAC along with SHA-2. The proposed processors are capable of computing both message digests and MACs. When computing MACs, the module is able to pre-process long keys, as well as to re-utilize previously used keys therefore achieving high throughput. When implemented on an Altera Stratix III FPGA, HMAC/SHA-224 uses 2262 ALUTs. If higher levels of security are demanded, HMAC/SHA-512 can be utilized, thus requiring 4601 ALUTs. Furthermore, the computation of a SHA-2 and HMAC operations over 1-block messages achieve throughput as high as 1.5Gbps and 760Mbps, respectively. Moreover, the proposed HMAC processor is energy efficient. For instance, no more than $5\mu J$ and $1.6\mu J$ per message block is consumed, respectively, when a new key is used and the key is reused.

In general, the proposed processors achieve higher efficiency in terms of throughput/area than third party implementations. These results show that SHA-2 with HMAC will not necessarily lead to slower implementations or higher energy consumption. Consequently, it is completely feasible to efficiently replace SHA-1 and MD5 with SHA-2 in hardware implementations of HMAC. These results are fundamental to support the implementation of higher levels of security in high-end and constrained applications such as servers and mobile devices.

8. REFERENCES

[1] Virtex-E 1.8V field programmable gate arrays. Data Sheet DS022-1 (v2.3), Xilinx Inc., July 2002.

[2] APEX 20K programmable logic device family. Data Sheet DS-APEX20K-5.1, Altera Corp., March 2004.

[3] Virtex-II platform FPGAs. Data Sheet DS031 (v3.5), Xilinx Inc., November 2007.

[4] Stratix III device handbook. Data Sheet SIII5V1-1.7, Altera Corp., February 2009.

[5] Altera Corp. *Avalon Interface Specifications*, April 2009.

[6] Altera Corp. *Nios II Processor Reference Handbook*, March 2009.

[7] Altera Corp. *QuartusII Handbook v9.0*, March 2009.

[8] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol. RFC 5246, IETF Network Working Group, August, 2008.

[9] I.Yiakoumis, M.Papadonikolakis, and H.Michail. Efficient small-sized implementation of the keyed-hash message authentication code. In *Proceedings of the IEEE Eurocon Conference 2005 Computer as a Tool*, volume 2, pages 1875–1878, November 2005.

[10] E. Kaufman. Internet key exchange (IKEv2) protocol. RFC 4306, IETF Network Working Group, December 2005.

[11] S. Kelly and S. Frankel. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. RFC 4868, IETF Network Working Group, May 2007.

[12] S. Kent. IP authentication header. RFC 4302, IETF Network Working Group, December 2005.

[13] S. Kent. IP encapsulating security payload (ESP). RFC 4303, IETF Network Working Group, December 2005.

[14] S. Kent. Security architecture for the internet protocol. RFC 4301, IETF Network Working Group, December 2005.

[15] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr. Design and performance analysis of a reconfigurable, unified HMAC-hash unit. *IEEE Transactions on Circuits and Systems I*, 54(12):2683–2695, December 2007.

[16] M. Kim, Y. Kim, J. Ryou, and S. Jun. Efficient implementation of the keyed-hash message authentication code based on SHA-1 algorithm for mobile trusted computing. In *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC 2007)*, pages 410–419, 2007.

[17] M. McLoone and J. McCanny. A single-chip IPSec cryptographic processor. pages 133–138, October 2002.

[18] H. Michail, A. Kakarountas, A. Milidonis, and C. Goutis. Efficient implementation of the keyed-hash message authentication code (HMAC) using the SHA-1 hash function. In *Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems*, pages 567–570, 2004.

[19] NIST. Advanced encryption standard (AES). Federal Information Processing Standards Publication FIPS PUB 197, November 2001.

[20] NIST. The keyed-hash message authentication code (HMAC). Federal Information Processing Standards Publication FIPS PUB 198, March 2002.

[21] NIST. Digital signature standard (DSS). Federal Information Processing Standards Publication Draft FIPS PUB 186-3, November 2008.

[22] NIST. Secure hash standard (SHS). Federal Information Processing Standards Publication FIPS PUB 180-3, October 2008.

[23] R. Rivest. The MD5 message-digest algorithm. RFC 1321, IETF Network Working Group, April 1992.

[24] G. Selimis, N. Sklavos, and O. Koufopavlou. VLSI implementation of the keyed-hash message authentication code for the wireless application protocol. In *Proceedings of 10th IEEE International Conference on Electronics, Circuits and Systems (ICECS'03)*, pages 24–27, December 2003.

[25] M.-Y. Wang, C.-P. Su, C.-T. Huang, and C.-W. Wu. An HMAC processor with integrated SHA-1 and MD5 algorithms. In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation (ASP-DAC'04)*, pages 456–458, 2004.

[26] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Proceedings of Crypto 2005*, pages 17–36, 2005.

[27] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT 2005*, pages 19–35, 2005.