Divide and Concatenate: A Scalable Hardware Architecture for Universal MAC

Bo Yang Ramesh Karri
ECE Department
Polytechnic University, Brooklyn, NY, 11201
yangbo@photon.poly.edu, ramesh@india.poly.edu

David A. McGrew
Cisco Systems, Inc.
San Jose, CA 95134
mcgrew@cisco.com

Abstract

We present a cryptographic architecture optimization technique called divide-and-concatenate based on two observations: (i) the area of a multiplier and associated data path decreases exponentially and their speeds increase linearly as their operand size is reduced. (ii) in hash functions, message authentication codes and related cryptographic algorithms, two functions are equivalent if they have the same collision probability property. In the proposed approach we **divide** a 2w-bit data path (with collision probability 2^{-2w}) into two w-bit data paths (each with collision probability 2^{-w}) and **concatenate** their results to construct an **equivalent** 2w-bit data path (with a collision probability 2^{-2w}).

We applied this technique on NH hash, a universal hash function that uses multiplications and additions. When compared to the 100% overhead associated with duplicating a straightforward 32-bit pipelined NH hash data path, the divide-and-concatenate approach yields a 94% increase in throughput with only 40% hardware overhead. The NH hash associated message authentication code UMAC architecture with collision probability 2^{-32} that uses four equivalent 8-bit divide-and-concatenate NH hash data paths yields a throughput of 79.2 Gbps with only 3840 FPGA slices when implemented on a Xilinx XC2VP7-7 Field Programmable Gate Array (FPGA).

1. Motivation

In the past, most cryptographic algorithms have been developed to run fast on general-purpose processors. More recently, dedicated cryptographic hardware is being developed and deployed to match the >10 Gbps wire speed requirements. In this paper we will investigate scalable hardware architectures for cryptographic algorithms.

Cryptographic algorithms such as block encryption and message authentication are iterative, data-driven algorithms. These algorithms take an input message and a user key and generate a result after several iterations. Since these cryptographic algorithms are data-dominated, their hardware implementations are data path dominated with only a small amount of control logic. Arithmetic operations such as add, multiply and shift/rotate are at the core of these cryptographic algorithms. One straightforward approach to speeding up cryptographic hardware is to use fast implementations of adders, multipliers and other components [13]. Orthogonal to the circuit level and logic level approaches are architectural level speed-up techniques such as pipelining and loop unrolling [11].

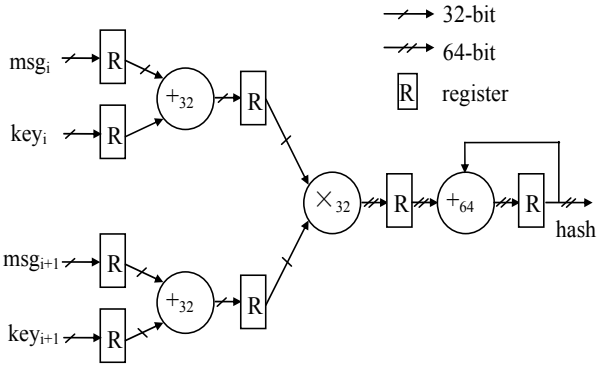

Figure 1: 32-bit data path for the NH hash function

Consider the NH hash data path shown in Figure 1 that is the core building block of the UMAC (Universal Message Authentication Code). This data path can be implemented as a three-stage pipeline. In the first stage, two 32-bit adders are used to add two adjacent message words to their corresponding sub keys. In the second stage, these two 32-bit intermediate results are multiplied. In the final stage the 64-bit result from a multiplier is accumulated into an output register using a 64-bit adder. We implemented this three-stage pipeline on a Xilinx Virtex II FPGA device using a single cycle adder and a single cycle multiplier. Since the clock rate of a 64-bit combinational adder is 193MHz and the clock rate of a 32-bit combinational multiplier is 83MHz, the throughput of this design is limited by the throughput of the multiplier and equals 5.3Gbps (=64bits×83MHz). Replacing the single cycle multiplier by a 5-stage 32-bit pipelined Xilinx multiplier core doubles the clock rate of the multiplier stage and that of the design to 160MHz. This in turn doubles the throughput of the design to 10.2Gbps (=64 bits×160 MHz). Finally, we can replicate this pipelined

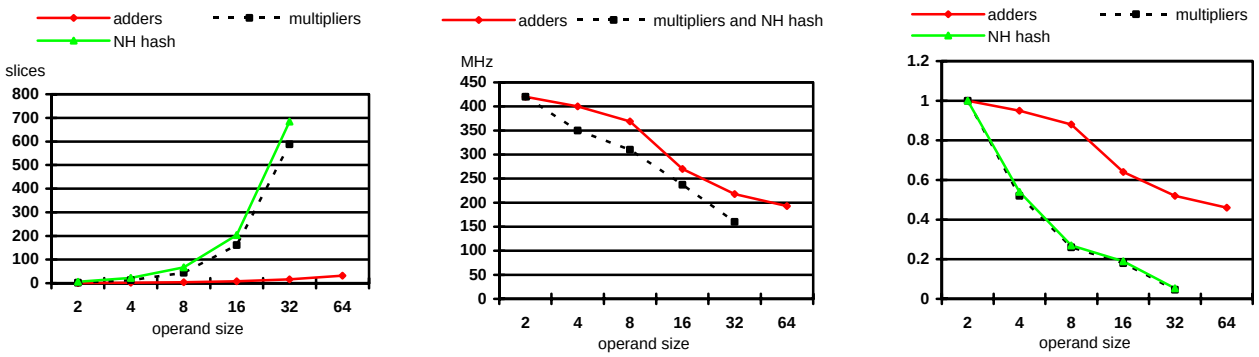

Figure 2: Area and throughput of adders, multipliers and NH hash data path as a function of operand size. Since (a) the area of a multiplier and NH hash data path grow exponentially and (b) their clock rates decrease linearly with operand size, (c) the throughput/area ratio of multiplier and NH hash data path decrease exponentially with operand size.

data path, with each copy operating on an independent input stream to get an additional improvement in throughput. However, this approach is expensive.

1.1. Can we do any better?

Cryptographic algorithms use large bit-width operations to improve security. For example, UMAC uses 32-bit and 64-bit additions and 32-bit multiplications. 128-bit Advanced Encryption Standard uses 32-bit additions and multiplications. Let us now analyze bottlenecks in speeding up wide operand data paths.

- The hardware-complexity of a multiplier and NH data path increase exponentially with operand size, while the hardware-complexity of an adder increases linearly with operand size. This is summarized in Figure 2(a). For example, while an 8-bit multiplier consumes 44 Virtex II slices, a 16-bit multiplier consumes 161 Virtex II slices and a 32-bit multiplier consumes 588 Virtex II slices. Furthermore, the area of an adder is much smaller than that of a multiplier for operand sizes ≥ 8-bits.
- From Figure 2 (b), it can be seen that a 2-bit adder and a 2-bit multiplier have identical speeds of 420MHz, the speed limit of the target FPGA device. This is because each output bit of 2-bit adder and multiplier is a 4-to-1 function and can be implemented exactly in one FPGA Look Up Table (LUT). Furthermore, the clock rates of adders and multipliers decrease almost linearly with the size of the operand. For example, the clock rates of 4-bit, 8-bit, 16-bit and 32-bit multipliers are 353MHz, 310MHz, 237MHz and 160MHz respectively. Since the clock rate of an adder is ≥ that of a multiplier with same operand size, multiplier delay determines the critical path of NH hash data path.
- Figure 2(c) shows that the normalized throughput/area ratio (using the ratio for a 2-bit unit as 1) for multipliers and the NH hash decreases exponentially with operand size. The case for adders is less dramatic.

In a nutshell, (1) crypto implementations that do not use multiplications are superior to those that do. (2) For a given operand size, adders yield more throughput (in Gbps) per unit area (per FPGA slice) than a multiplier. (3) Implementations that use small sized operands are superior to those that use large operands.

In this paper we propose to **divide** a 2w-bit data path into two w-bit data paths and **concatenate** their results to construct an **equivalent** 2w-bit data path. The concept of equivalence is crucial. Obviously, a straightforward data path and the corresponding divide-and-concatenate data path cannot be equivalent in terms of the results that they output. We define two data paths to be equivalent if the results that they output satisfy a pre-defined property. For cryptographic algorithms such as hash functions and message authentication codes the actual result is not important. Rather, it is the collision probability of the result that is important. Hence, we propose that two data paths implementing a hash function be considered equivalent if they have the same collision probability.

In the rest of this paper we will describe the divide-and-concatenate architecture optimization technique. Specifically, we will introduce universal hash functions, NH hash a universal hash function and the UMAC message authentication code based on NH hash in section 2. We will apply the divide-and-concatenate technique to NH hash in section 3. In section 4 we will present the architecture for the UMAC based on the NH hash architectures from section 3. We will then compare our work with related work in section 5 and finally summarize our contributions in section 6.

2. Universal Message Authentication Code

UMAC is a NESSIE (New European Schemes for Signatures, Integrity, and Encryption) message

authentication code standard [16]. The core of UMAC is the universal hash function NH hash. In this section we will describe universal hash functions in general and the NH hash in particular.

2.1. Universal Hash Function

Carter and Wegman [4] defined a universal hash function as follows: Let A and B be two sets, and let H be a family of functions from A to B. H is a universal family of hash functions if for every pair x1, x2 $\in$ A with x1$\neq$x2, and h(x1), h(x2) $\in$ B, the collision probability of h(x1) = h(x2) equals to 1/|B| and h in H. |B| is size of set B and 1/|B| is the smallest possible value of the probability. When B is small, the collision probability is large.

A MAC that uses a universal hash function as a building block hashes the input message M down to a small-size hash value using the universal hash function and then applies a cryptographic primitive to this hash value [5]. Since universal hash functions can compress the message M efficiently, the associated MACs are fast. Several universal hash functions and associated MACs have been proposed including MMH [6], Square Hash [7], bucket-hash [8], TMMH [9] and NH hash [3]. MACs are used by a receiver to verify whether the data received from the sender is not modified by a third party during transmission by computing the MAC of the received message using the secret key shared with the sender and matching it with the received MAC.

2.2. NH Hash

NH hash is a universal hash function that uses additions and multiplications; the operations correspond to machine instructions on modern processors. When NH hash is implemented on a modern processor, it can calculate the hash value for a 1024 word (a word is 32-bit wide) message using 1024 32-bit word sub keys as follows:

$(M_1 +_{32} K_1) \times (M_2 +_{32} K_2) +_{64} \ldots +_{64} (M_{k-1} +_{32} K_{k-1}) \times (M_k +_{32} K_k)$

M_i and K_i are 32-bit message words and corresponding sub key words. $+_{32}$ and $+_{64}$ are addition mod 2^{32} and addition mod 2^{64} respectively. The fastest implementation will require (i) 1024 32-bit adders to add each message word with its corresponding sub key word in the first step; (ii) 512 32-bit multipliers to multiply adjacent pairs of results from the addition step and (iii) a 9-level balanced addition tree composed of 511 64-bit adders to generate the hash value. This is expensive and in normal applications data paths are much smaller.

A 32-bit NH hash data path that operates on 32-bit input message words and 32-bit sub key words has a collision probability of 2^{-32}. Generally, A w-bit NH hash data path that operates on w-bit input message words and w-bit sub key words has a collision probability of 2^{-w} [3].

2.3. Reducing Collision Probability of NH hash

Since the bit-width w of the NH hash function is determined by the architecture of the underlying processor, increasing w is not always a feasible solution to reduce the collision probability. However, since NH is a universal hash function, its collision probability of 2^{-w} can be reduced to 2^{-nw} by hashing the same message n times using n independent keys and concatenating the results [5]. For example, if we hash a message twice using the w-bit NH hash function, each time with a different set of sub keys and concatenating the two hash values, the collision probability will drop from 2^{-w} to 2^{-2w}.

However this solution requires 2× key material. The Toeplitz-extension described in [3] reduces the amount of sub key material making this approach practical. As shown in Figure 3, when we use two 16-bit data paths to construct a 32-bit NH hash using Toeplitz-extension, the sub keys for the second data path are obtained by shifting the corresponding sub keys of the of the first data path using component S. When Toeplitz-extension is used a single 1024 w-bit-word sub key RAM is sufficient independent of the number of data paths.

3. Divide and Concatenate: An Architecture Level Optimization Technique

The straightforward 32-bit NH hash data path shown in Figure 1 takes two 32-bit message words every cycle and generates a 64-bit hash value after the entire message is processed.

Using the divide-and-concatenate technique a 32-bit NH hash data path with a collision probability of 2^{-32} can be constructed using two 16-bit NH hash data paths, each with collision probability of 2^{-16}, and concatenating their 32-bit results to generate a 64-bit hash value. This corresponds to the two data paths at the top of Figure 3.

Using the data from Figure 2, the throughput of the straightforward 32-bit NH hash data path is 10.24 Gbps (=160MHz×64bits), while the throughput of the equivalent 16-bit divide-and concatenate data path is 7.58 Gbps (=237MHz×32bits). The equivalent 16-bit divide-and-concatenate data path consumes 408 FPGA slices (=204 slices per 16-bit NH hash data path×2) compared to 684 slices by the straightforward 32-bit data path. Overall, the throughput/area ratio of 0.0186 Gbps/slice (=7.58 Gbps ÷ 408 slices) for the divide-and-concatenate architecture is 25% more efficient than 0.0149 Gbps/slice (=10.24 Gbps ÷ 684 slices) for the straightforward architecture.

The duplicated divide-and-concatenate architecture shown in Figure 3 uses four 16-bit NH hash data paths and processes a 64-bit input every cycle (same as the straightforward 32-bit NH data path) yielding a throughput of 15.17 Gbps (=237MHz×64bits) with an area of 816 slices (=204slices×4units). The fixed shift operation S and

the concatenation operation || in Figure 3 do not contribute to the area overhead in hardware. Compared to the straightforward 32-bit NH data path, this data path yields a 48% (=15.17Gbps÷10.24Gbps) improvement in throughput with an associated area overhead of 19% (=816slices÷684slices).

Let us apply this divide-and-concatenate technique once more and construct each 16-bit NH hash data path using four 8-bit NH hash data paths. This translates into sixteen 8-bit NH hash data paths to construct an equivalent 32-bit NH hash data path. From Figure 2, the area of an 8-bit NH hash data path is 60 slices and the clock rate is 310 MHz, yielding a throughput of 19.84 Gbps (=310 MHz×64 bits). The area for this equivalent 8-bit NH hash data path is 960 slices (=60 slices ×16 units). Compared to the straightforward 32-bit NH hash data path, this 8-bit equivalent data path yields a 94% improvement in throughput (=19.84Gbps÷10.24Gbps) with an area overhead of 40% (=960slices÷684slices). The throughput/area ratio of this 8-bit data path is 0.021 Gbps/slice (19.84Gpbs÷960slices), greater than 0.0186 Gbps/slice of the 16-bit data path based architecture.

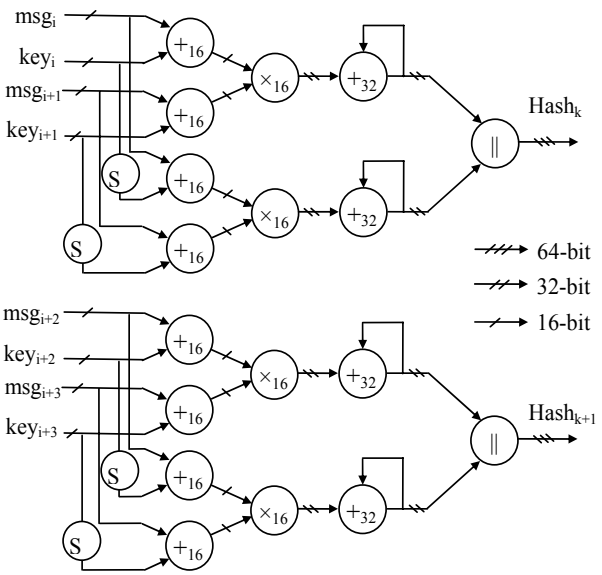

Figure 3: This NH data path composed of four 16-bit NH hash data paths has a collision probability of 2^{-32}. It processes two 32-bit messages per clock cycle in parallel and generates two 64-bit hash values.

Does this mean that the more components that the basic data path is divided into, the better the resulting divide-and-concatenate data path? Let us construct a 32-bit NH hash data path using 64 4-bit data paths. The throughput of this data path is 22.6Gbps (=353MHz×64bits) and area is 1280 slices (=20slices×64units) yielding a throughput/area ratio of 0.0177 Gbps/slices (22.6 Gpbs÷1280slices, which is less than that for the 8-bit equivalent NH hash data path.

Figure 4 summarizes the area, throughput and throughput/area ratio for five equivalent 32-bit NH hash data paths (i.e., all these data paths have a collision probability of 2^{-32} and process 64 input bits every clock cycle). The 8-bit data path has the best throughput/area ratio. It achieves 90% throughput improvement with only 40% area overhead. Compare this with the 100% area overhead when the straightforward 32-bit data path is duplicated achieve a 100% improvement in throughput.

4-bit and 2-bit data path based designs are not as efficient as an 8-bit data path based design. In the divide-and-concatenate approach, the number of adders increases exponentially while their area decreases linearly. In 8-bit and larger designs the NH hash area is dominated by large multipliers. On the other hand, in 4-bit and 2-bit data path based designs the NH area is dominated by adders. This is because the area of an adder is comparable to that of a multiplier and the number of adders grows exponentially.

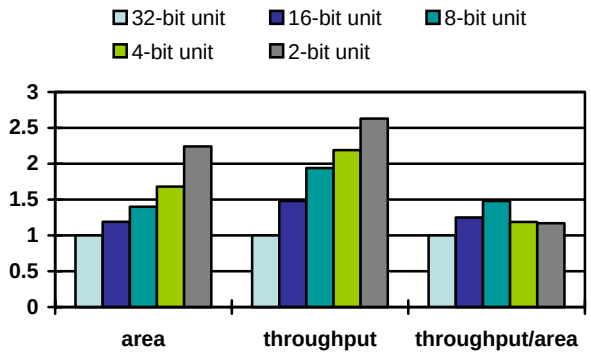

Figure 4: Normalized area, throughput and throughput/area ratio of five equivalent divided-and-concatenated data paths for a NH hash with collision probability of 2^{-32}.

A drawback of the divide-and-concatenate technique is that the length of the output hash doubles as you go from the straightforward 32-bit data path to an equivalent 16-bit data path to an equivalent 8-bit data path and so on. In fact, the length of hash value of an equivalent 2-bit data path is 16 times longer than that for the straightforward data path. We will discuss how this increase in output length impacts the overall performance of UMAC in next section.

4. Putting It All Together: The UMAC Architecture

The Universal Message Authentication Code has three steps:

- Step 1: sub key generation:

In this step, the user key is expanded into (i) 1024 w-bit-word sub keys that are used by NH hash in step 2 and (ii) 512-bit key A for the HMAC-SHA1 cryptographic primitive used in step 3. Since sub keys are generated just once at the beginning of a session, this step does not impact the overall throughput of UMAC.

- Step2: Hashing the input message using NH hash:

If the message is larger than 1024 w-bit word, it will be partitioned into 1024 w-bit word blocks. Each block is compressed independently according to the method explained in section 2.2 and the final hash value is obtained as the concatenation of the hash values of each block and the message length encoded in binary:

HM = NH(Msg blk 1)||NH(Msg blk 2)…||NH (Msg blk t)|| Message Length

- Step 3: Computing the MAC:

HMAC-SHA1, a cryptographic MAC, is applied to the concatenated hash HM from step 2 to obtain fixed length MAC.

$MAC = HMAC\text{-}SHA1_A (HM \,||Nonce)$

Typically the nonce is a counter which the sender increments with each transmitted message.

HMAC-SHA1 operates on 512-bit block and generates a 160-bit result after 80 cycles. This 160-bit result is used by HMAC-SHA1 when it processes the next 512-bit message block. When all message blocks are consumed, this 160-bit result is the MAC. The fastest implementation of HMAC-SHA1 on an FPGA yields a throughput of 652 Mbps and consumes 569 slices [10].

When the UMAC architecture processes a single message stream, multiple NH data paths hash different parts of the message and concatenate them into HM as described in step 2. HM is then hashed into a MAC using HMAC-SHA1. The divide-and-concatenate approach cannot be applied to HMAC-SHA1 as it is not a universal hash function.

In general, the throughput of HMAC-SHA1 is not a bottleneck for the overall UMAC architecture. The straightforward 32-bit NH hash data path compresses each 1024×32-bit message block input into a 64-bit intermediate value or hash value output yielding a compression ratio of 512. The throughput of this straightforward NH hash data path is 10.24 Gbps at its input and 20 Mbps (=10.24 Gbps÷512) at its output (which is also the input to HMAC-SHA1). This is 32 times smaller than the 652 Mbps throughput of HMAC-SHA1. The 2-bit equivalent NH hash data path generates an output of 1024 bits translating into a compression ratio of only 32 (=1024×32-bit input ÷1024-bit output). The resulting throughput of 840Mbps (=26.9 Gbps÷32) at the input to the HMAC-SHA1 is > 652 Mbps throughput of HMAC-SHA1. Now, HMAC-SHA1 is the bottleneck. Table 1 summarizes the throughput, compression ratio and output data rate of the equivalent data paths for the 32-bit NH hash.

Table 1: Input throughput, compression ratio, output data rate for equivalent architectures of a 32-bit NH hash.

Size	2bit	4bit	8bit	16bit	32bit
Thro'put (Gbps)	26.9	22.6	19.8	15.2	10.2
Comp. ratio	32	64	128	256	512
O/p data rate (Mbps)	840	353	155	60	20

To effectively utilize HMAC-SHA1 data path, the UMAC architecture can be designed such that HMAC-SHA1 works with multiple message streams and associated NH hash data paths. For example, since the data rate at the output of straightforward 32-bit NH hash architecture is 20Mbps and the throughput of HMAC-SHA1 is 654 Mbps, one HMAC-SHA1 data path can work with thirty two 32-bit NH hash data paths and associated message streams. This translates into an effective UMAC throughput of 326.4 Gbps. Table 2 summarizes the maximum throughput, maximum number of independent message streams that can be supported, area (NH hash data path + HMAC-SHA1 data path) and throughput/area ratio of five UMAC architectures. For example, for 8-bit equivalent UMAC architecture, we use 4(=654Mbps÷155Mbps) 16-bit NH hash data paths with 960 slices. Its area is 3840 slices (=960slices per equivalent NH hash×4+569slices for HMAC-SHA1). Its throughput is 79.2 Gbps (19.8 Gbps per equivalent NH hash data path× 4). The 8-bit equivalent architecture has the best throughput/area ratio.

Table 2: Effective throughput, maximum number of independent message streams that can be supported, area (NH hash data path + HMAC-SHA1 data path) and throughput/area ratio of four UMAC architectures.

Architecture	4-bit	8-bit	16-bit	32-bit
Max thro'put (Gbps)	22.6	79.2	167.2	326.4
Area(slices)	1849	3840	9545	22457
Throughput/Area (Gbps/slice)	0.012	0.021	0.018	0.015

5. Comparison to Related Work

Early MAC algorithms used private-key based block ciphers in the cipher block-chaining (CBC) mode [1]. The hardware implementation of most block ciphers produces moderate throughput. The throughput of FPGA implementation of RC6 varies from 88.5 Mbps with 2638 slices to 2.397 Gbps with 10856 slices [11]. Many academic and commercial IP cores of AES are available. The most efficient implementation has the throughput of 1.19 Gbps with 450 slices and 10 Block RAMs [12].

The second class of MACs uses cryptographic hash functions such as MD5 and SHA1. MD5 and SHA1 are IETF MAC standards. Krawczyk et al. [2] defined HMAC by combining traditional hash functions with keying. A commercial implementation of MD5 has a throughput of 528 Mbps and area cost of 551 slices and 1 BlockRAMs [17].

Most universal hash functions use modular multiplication [6] [7] [8] [9] [3] and hence the throughput of associated MAC depends on the speed of modular multiplication [14] [15]. The divide-and-concatenate technique can be applied to hardware implementation of this class of MACs.

6. Conclusions

Applying general hardware design techniques to cryptographic architectures yields only moderate improvements. We defined a collision probability equivalent data path and combined it with the divide-and-concatenate technique to design efficient high throughput architectures. We characterized the area and throughput of equivalent data paths of 32-bit NH hash and demonstrated that the 8-bit equivalent NH hash data path is the most efficient architecture with the associated UMAC architecture achieving 79.2 Gbps using only 3840 FPGA slices.

The divide-and-concatenate technique cannot speed-up software implementations but can only improve the collision probability beyond that provided by the processor architecture. This is because, if a processor supports w-bit additions and multiplications in one or two cycles, then w/2-bit operations will also consume the same number of cycles as w-bit operations.

References

[1] ISO/IEC 9797-1. Information technology - security techniques - data integrity mechanism using a cryptographic check function employing a block cipher algorithm. International Organization for Standards, Geneva, Switzerland, Second edition, 1999.

[2] M. Bellare, R. Canetti, H. Krawczyk. Keying hash functions for message authentication. In Advances in Cryptology - CRYPTO'96 vol. 1109 of Lecture Notes in Computer Science Springer-Verlag pp. 1-15, 1996.

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. Wiener, editor, Advances in Cryptology – CRYPTO '99, volume 1666 of Lecture Notes in Computer Science, pp. 216–233. Springer-Verlag, Berlin Germany, Aug.1999.

[4] L. Carter, and M. Wegman. Universal hash functions. Journal of Computer and System Sciences, Vol 18, pp. 143–154, 1979.

[5] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. Journal of Computer and System Sciences, vol. 22, pp. 265–279, 1981.

[6] S. Halevi, and H. Krawczyk. MMH: Software message authentication in the Gbit/second rates. In Proceedings of the 4th Workshop on Fast Software Encryption, vol. 1267, Springer-Verlag, pp. 172-189, 1997

[7] M. Etzel, S. Patel, and Z. Ramzan. Square hash: Fast message authentication via optimized universal hash functions, Advances in Cryptology, volume 1666 of Lecture Notes in Computer Science, pp. 234-251, 1999.

[8] T. Johansson. Bucket hashing with small key size. In Advances in Cryptology-Eurocrypt'97, Lecture Notes in Computer Science, Springer-Verlag, pp. 149-162, 1997

[9] D. A. McGrew. The Truncated Multi-Modular Hash Function (TMMH), IETF Internet Draft, 2001. http://www.mindspring.com/~dmcgrew/draft-mcgrew-saag-tmmh-01.txt

[10] Helion Technology. Datasheet-High Performance SHA1 Hash Core for Xilinx FPGA, 2003. http://www.heliontech.com/downloads/sha1_xilinx_helioncore.pdf

[11] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 9(4), pp. 545-557, August 2001.

[12] Helion Technologies. High Performance AES(Rijndael) cores for Xilinx FPGA, 2003. http://www.heliontech.com/downloads/aes_xilinx_helioncore.pdf

[13] C. K. Koc. RSA hardware implementation. Technical Report TR 801, RSA Laboratories, April 1996. http://islab.oregonstate.edu/koc/papers/r02rsahw.pdf

[14] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography, in Proceedings 11th Symposium on Computer Arithmetic, pp. 252-259, 1993.

[15] A. F. Tenca and C. K. Koc. A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm. IEEE Transactions on Computer, Vol. 52, No.9, pp. 1215-1221, September, 2003

[16] Nessie. Nessie Project Announces Final Selection of Crypto Algorithms, 2003. https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/press_release_feb27.pdf

[17] Helion Technology. Datasheet-High Performance MD5 Hash Core for Xilinx FPGA, 2003. http://www.heliontech.com/downloads/md5_xilinx_helioncore.pdf