Efficient Key-Dependent Message Authentication in Reconfigurable Hardware

Jérémie Crenne*, Pascal Cotret*, Guy Gogniat*, Russell Tessier†, and Jean-Philippe Diguet*
*Laboratoire Lab-STICC, Université de Bretagne-Sud, Lorient, France
†Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, USA

***Abstract*—Cryptographic message authentication is a growing need for FPGA-based embedded systems. In this paper a customized FPGA implementation of a GHASH function that is used in AES-GCM, a widely-used message authentication protocol, is described. The implementation limits GHASH logic utilization by specializing the hardware implementation on a per-key basis. The implemented module can generate a 128-bit message authentication code in both pipelined and unpipelined versions. The pipelined GHASH version achieves an authentication throughput of more than 14 Gbit/s on a Spartan-3 FPGA and 292 Gbit/s on a Virtex-6 device. To promote adoption in the field, the complete source code for this work has been made publically-available.**

I. Introduction

As the application space of FPGA systems continues to diversify, the importance of optimized high performance security solutions has grown in importance. For example, secure point-to-peripheral broadband communications now require data rates between 1 and 100 Gbit/s [1]. Often, the distributed nature of these channels makes them vulnerable to security attacks, necessitating low-overhead preventive measures. FPGA-optimized security blocks targeted to message authentication protocols are needed to reach this goal.

Recently, the block cipher Advanced Encryption Standard (AES) in counter mode (CTR) was combined with Galois Counter Mode (GCM) of operation [2] to provide both message encryption and authentication. This approach has proven popular since it is not constrained by intellectual property rights and has been shown to be provably secure [3]. A key aspect of GCM is a 128-bit Galois field multiplication GF(2^{128}). One or more instantiations of this GMULT operation are needed to perform the Galois Hash (GHASH) function required for message authentication. Mathematically, a cryptographic GHASH function is a construct that performs universal hashing over a binary Galois field to generate a message authentication code (MAC) [4]. The goal of the function is to authenticate the source of a message and its integrity. Although hardware and software implementations of GHASH are available [5], most require the use of multipliers which make them less suitable for low-cost, resource-efficient FPGAs.

978-1-4577-1740-6/11/$26.00 © 2011 IEEE

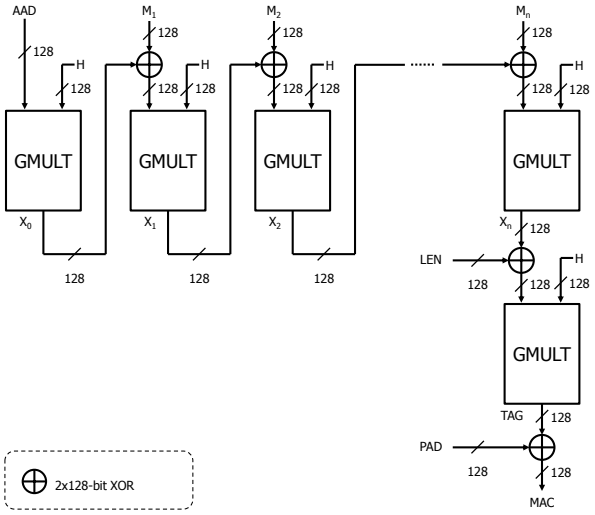

Fig. 1. Block diagram of a combinational implementation of the GHASH function.

In this work, a GHASH implementation customized for FPGA deployment is described. The GHASH module is specifically designed to take advantage of the specialization offered by FPGA lookup tables and flip flops. The custom key used for authentication is synthesized into the module structure, specializing the associated circuitry and reducing module area. A pipelined version of the module is presented to provide high throughput. Message authentication rates of up to 292 Gbit/s have been evaluated.

The rest of this paper is organized as follows. Background on AES-GCM and GHASH functions is provided in Section II. Implementation details of our approach are provided in Section III and experimental results are discussed in Section IV. Section V concludes the paper.

II. Background

A. GHASH Functional Definition

As shown in Figure 1, a GHASH function is composed of chained GF(2^{128}) multipliers (GMULT) and bitwise exclusive-OR (XOR) operations. Each 2x128 XOR function includes 128 2-bit XOR operations.

GHASH inputs include: 1) A 128-bit hash key H. This key is derived from a symmetric cryptographic key K. 2) An M-bit message requiring authentication. The message can be divided into n 128-bit blocks M_1-M_n. If necessary, the last message block M_n is padded with zeros to create a 128-bit word. 3) An optional 128-bit additional authenticated data (AAD) value. This data value, which is authenticated but not encrypted, is generally used to identify the source of an authenticated message. 4) A 128-bit LEN value which expresses the word lengths of AAD and the message M. 5) A 128-bit cryptographic pad value (PAD) which ciphers the function output TAG to generate the message authentication code (MAC). The resulting 128-bit is expressed as:

$$
\begin{aligned}
H &= E(K, 0^{128}) && (1)\\
X_0 &= GMULT(H, AAD) && (2)\\
X_i &= GMULT(H, M_i \oplus X_{i-1}) && (3)\\
&\quad for\ i = 1...n\\
LEN &= length(AAD)_{64} \mid\mid length(M)_{64} && (4)\\
TAG &= GMULT(H, X_n \oplus LEN) && (5)\\
MAC &= PAD \oplus TAG && (6)
\end{aligned}
$$

where $E(K, B)$ denotes an AES block cipher encryption of a value B with a secret key K. The expression 0^{128} denotes a string of 128 zero bits, and $A \parallel B$ denotes the concatenation of bit strings A and B. The multiplication of two elements $A, B \in$ GF(2^{128}) is denoted as GMULT(A,B), and the addition in the Galois field of A and B, denoted as $A \oplus B$, is equivalent to the XOR operation. The function $length()$ returns a 64-bit word describing the number of bits in its argument, with the least significant bit on the right. In general, an efficient GHASH implementation depends on the software or hardware design of the GF(2^{128}) multiplier.

B. GHASH Hardware Implementation

Previously, Paar [6] summarized the efficiency of various hardware finite field multiplication methods for GF(2^q). Although bit serial implementations have linear area and performance with $O(q)$ and digit serial implementations with digit size D vary in area with $O(qD)$ and performance as $O(q/D)$, their performances are generally considered insufficient for contemporary message authentication throughput.

Even though the sizes of parallel implementations are generally larger than serial- and digit-based implementations, the desired throughput performance of authentication motivates their use. Since the hardware complexity of a parallel implementation is $O(q^2)$, a 128-bit implementation which includes over 10,000 lookup tables (LUTs) can easily be required if hardware optimizations are not considered. Fortunately, multiplication over GF(2^{128}) can be expressed as a series of polynomial multiplications and modular reductions, leading to implementations based on Reyhani-Masoleh [7], Mastrovito [5] and Karatsuba-Ofman algorithms. These implementations have been shown to provide multi-Gbit/s throughput at the cost of over 8,000 LUTs per function. In Section IV, a comparison of our new GHASH implementations including an optimized GF(2^{128}) multiplier is made versus these previous approaches.

C. GHASH Software Implementation

Ideas for FPGA-optimized hardware implementations of GF multiplication in GHASH functions can be identified by considering previous software implementations. Software binary field multiplication generally uses a variety of time-memory tradeoffs [8]. Currently, software implementations take two forms, one which considers the hash key H from Equation (1) as fixed and one which assumes a time-changing H value. The multiplication operation GMULT(H,B) between the hash key H and an arbitrary element B, as shown in Equations (2), (3) and (5), is linear over the field GF(2). By setting H to be constant, this property can be exploited to allow efficient table-driven lookups for function results rather than expensive GF operations [8]. In many cases, the table-driven approach provides a significant software performance improvement for a modest memory cost. Table implementation can be optimized to limit the amount of memory required to encode operations based on H and allow multiport (parallel) table access to provide high throughput.

D. New Hardware Implementation and Limitations

In our new hardware module implementation, constant key specialization in the FPGA is used. The fine-grained LUT parallelism found in FPGAs is used to implement a precomputed table for GMULT operations based on a constant H. As shown in Section IV, this provides the implementation of a parallel GF(2^{128}) multiplier with significantly reduced LUT count while providing multi-Gbit/s throughput. The specialization of the GMULT table can be accommodated for multiple keys if a portfolio of bitstreams for different H values is maintained. These benefits can have a direct impact on some, but not all, applications of AES-GCM.

Network attacks are a concern for a variety of organizations. Preventing the unauthorized access, modification, and misuse of network resources is key to providing a secure environment. Virtual private networks (VPN) are widely employed to connect private local area networks to remote locations. Each connection uses a secure tunnel over an unsecure channel for packet transmission. For many VPNs, the secret key used for encryption and authentication is changed on

Algorithm 1 *Precomputation of the table* T; $H \in GF(2^{128})$

1: $R \leftarrow 11100001 \parallel 0^{120}$
2: $H \leftarrow E(K, 0^{128})$
3: $V \leftarrow H$
4: **for** $i = 0$ to 127 **do**
5: **if** $V_{127} = 0$ **then**
6: $V \leftarrow rightshift(V)$
7: **else**
8: $V \leftarrow rightshift(V) \oplus R$
9: **end if**
10: $T[i] \leftarrow V$
11: **end for**

a weekly, monthly or yearly basis. Current commercial high-end security appliances allow a maximum throughput of 40 Gbit/s and potentially up to 10,000 client VPN users per session [9]. VPN infrastructure can potentially benefit from our key-dependent GHASH implementation to achieve a throughput of 40 Gbit/s. Another application which requires authentication with slow changing keys is embedded system memory protection [10]. This application requires infrequent key changes over weeks or months. Additional applications with infrequent key changes could similarly benefit from key specialization.

As mentioned earlier, not all AES-GCM applications are suitable for our approach. The IEEE MACSEC Ethernet encryption standard uses AES-GCM for authentication [11]. In typical use, the required key may change on a per-packet basis. In our implementation this would require per-packet FPGA reconfiguration, a prohibitive cost.

III. GHASH Design Architectures

In this section, a GHASH module which generates authenticated 128-bit data every two clock cycles is described. This design combines the two combinatorial GMULT blocks shown on the left of Figure 1 with an output register. This module is designed to be easily integrated into a complex design. Multiple instantiations of the module can be chained together to form a higher throughput pipelined implementation.

A. Table precomputation

The efficient use of a GMULT lookup table based on a fixed H requires the precalculation of table values. For a constant value H, a table T can be constructed to represent GF(2^{128}) multiplication between an H value and a 128-bit input value. Each element of the table is a 128-bit vector. Based on GMULT specifications [2], the algorithm needed to fill the table is shown in Algorithm 1. The i^{th} bit of an element A is denoted as A_i. The leftmost bit is A_0, and the rightmost bit is A_{127}. The multiplication operation uses the

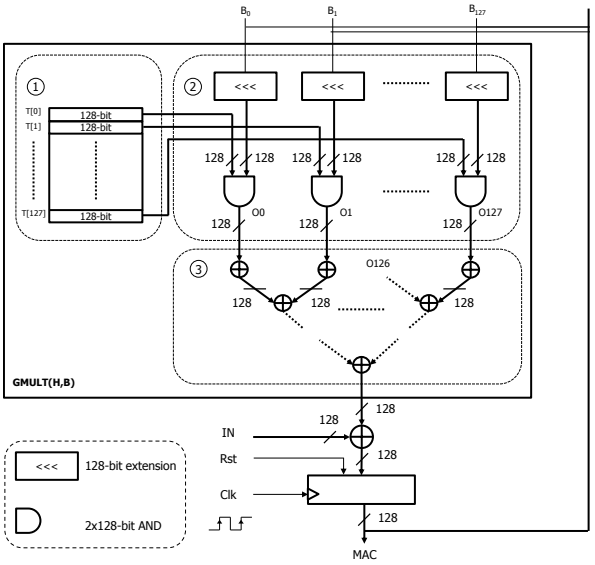

Fig. 2. Basic GHASH module implementation

element $R = 11100001 \parallel 0^{120}$ [2]. The function $rightshift()$ moves the bits of its argument one bit to the right. The hash key H is the result of the invocation of the AES block cipher with a zeroed 128-bit word at its input. This operation is highlighted in line 2 in Algorithm 1 where K is a secret key. Every 128-bit vector T is calculated with a simple conditional statement along with 128-bit XORs and right shifts (line 5 to 9).

The unoptimized table contains 128 vectors of 128 bits, i.e. 16,384 memory bits, either stored in registers or in RAM. As explained in the next subsection, a large fraction of the bits are needed in parallel. Direct table implementation in LUTs is clearly prohibitive due to size and performance concerns of implementation in more than 10,000 LUTs. Direct implementation in block RAMs (BRAMs) would at first appear to be a reasonable option since these primitives operate at high speed (e.g. 550 MHz on a Virtex-5). However, BRAMs have at most two read ports, limiting parallel access. To be able to perform needed parallel access to the data, the block RAM contents would need to be replicated numerous times (e.g. up to 64x for a Spartan device). In general, many FPGA designs are constrained by BRAM availability.

Our implementation avoids the distributed and parallel RAM problem by directly synthesizing binary 1 values in the table T into GMULT logic. For most H and associated K values, the bit value population of T is not strictly 50% 1s and 50% 0s, leading to possible optimization. As a result, the AND gate outputs shown in Figure 2 can be set to 0 in many cases, reducing the amount of logic required to implement the GHASH function. Fixed binary 1 AND inputs in the figure convert the AND gates accordingly. Subsequently,

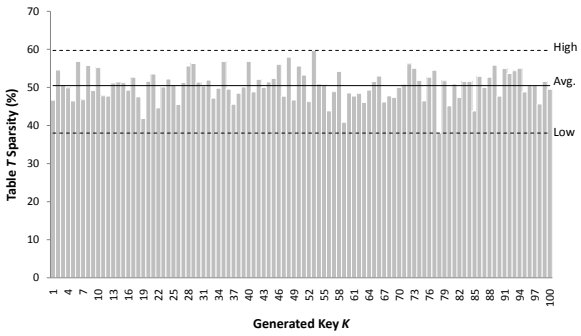

Fig. 3. Fraction of table T logic value 0s for 100 generated keys K

Algorithm 2 *Multiplication in* $GF(2^{128})$. *Computes the value* $X = GMULT(H, B)$ *with* H *constant;* B, $X \in GF(2^{128})$

1: **for** $i = 0$ to 127 **do**
2: $X \leftarrow 0$
3: **if** $B_i = 1$ **then**
4: $X \leftarrow X \oplus T[i]$
5: **end if**
6: **end for**
7: **return** X

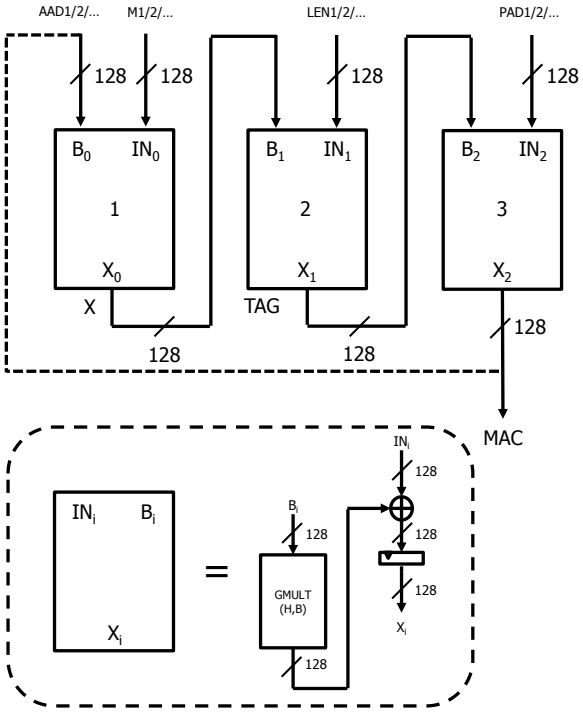

Fig. 4. A multi-stage pipelined GHASH module implementation. The bottom of the figure shows the structure of one pipeline stage

the 128×128-bit XOR tree can be pruned to further reduce logic. By trimming single bits with a 0 logic value and optimizing the 1 logic values before the mapping process, the synthesized design is reduced. Additionally, the logic is structured to take advantage of the wide-input, single output structure of FPGA LUTs.

To study the average logic value distribution of a table T, we randomly generated 10^8 keys K and evaluated the bit value 0 population of tables. The Mersenne Twister pseudo random number generation algorithm [12] was chosen to generate the K values. It has a period of 2^{19937}-1, is uniformly distributed, and passes numerous tests for statistical randomness, including the Diehard tests [13] and part of the stringent TestU01 tests [14]. For this given range of keys K, a low, average, and high logic value 0 percentage of 30%, 50%, and 70% was determined. A sample distribution for 100 keys K is shown in Figure 3. The logic density of a design is related to the percentage of bit value 0s. The tiniest and fastest module is obtained for a percentage of 70%. A 30% distribution results in a larger and slower implementation. Our designs implemented in Section IV assume a 50% distribution.

B. Basic module

The basic unoptimized GHASH module shown in Figure 2 contains a purely combinatorial GF(2^{128}) multiplier GMULT(H,B), a 128×128-bit XOR tree and a 128-bit output register. The detailed implementation of the multiplier (Algorithm 2) is the basis for this architecture. Three sub-modules form the basis for the design (Figure 2): 1) the 128×128-bit table T on the left (synthesized into logic), 2) the circuitry to perform conditional statements on the right and 3) the 128×128-bit XOR on the bottom right. To avoid a 1 cycle penalty during typical synchronous conditional statements, the statements are implemented in a combinatorial fashion. Every B input bit, B_0 to B_{127}, is extended 128 times (essentially a wire fanout) and ANDed with the corresponding vectors of table T. The module accepts a single 128-bit input B and generates one 128-bit output MAC. A feedback path allows iteration across multiple 128-bit chunks of a message for authentication, as shown in Figure 1. The operation of the module is controlled by a small sequencer. For example, to authenticate a message composed of a 128-bit AAD and a 512-bit message M, the input B follows the sequence AAD, M_1, M_2, M_3, M_4, LEN, and PAD. The resulting MAC is available in the output register after 6 cycles.

C. Pipelined module

A multistage pipelined module can be derived from multiple instantiations of the basic module. The basic design is replicated n times to provide a n-stage pipelined module. For every new stage, a GF(2^{128}) GMULT(H,B), a 2×128-bit XOR and a 128-bit register is added. Figure 4 illustrates the architecture

Design	Device				Resources			Frequency	Throughput	T-put/slice
	Family	Part	M / G / K	PAR	LUT	FF	Slices	(MHz)	$(Gbit/s)$	$(\frac{Mbit}{s \times slice})$
Basic GHASH	Virtex-4	xc4vlx25	○ / • / •	•	2,469	165	1,277	250.0	16.0	12.5
Basic GHASH	Virtex-5	xc5vlx50t	○ / • / •	•	2,008	128	533	303.0	19.4	36.4
Basic GHASH	Virtex-6	xc6vlx75t	○ / • / •	•	1,714	143	447	384.6	24.6	55.1
Basic GHASH	Spartan-3	xc3s1000l	○ / • / •	•	2,429	150	1,242	114.9	7.4	5.9
Pipe GHASH (n=4)[1]	Virtex-4	xc4vlx25	○ / • / •	•	12,186	793	6,182	222.2	113.8	18.4
Pipe GHASH (n=8)[1]	Virtex-5	xc5vlx50t	○ / • / •	•	17,611	1,152	4,594	232.6	238.1	51.8
Pipe GHASH (n=8)[1]	Virtex-6	xc6vlx75t	○ / • / •	•	15,414	1,358	3,985	285.7	292.6	73.4
Pipe GHASH (n=1)[1]	Spartan-3	xc3s1000l	○ / • / •	•	4,875	280	2,484	116.3	14.9	6.0
Huo *et al.* [7]	Virtex-5	xc5vlx30	• / ○ / ○	○	8,864	411	2,992[2]	240.2	30.8	10.3[2]
Lu *et al.* [15]	Virtex-5	xc5vlx50t	○ / • / ○	○	9,405	430	3,175[2]	120.2	15.4	4.8[2]
Zhou *et al.* [16]	Virtex-5	xc5vlx85	• / ○ / ○	•	*n/a*	*n/a*	4,628	324.0	41.5	8.9
Chen *et al.* [17]	Virtex-4	xc4vlx60	○ / • / ○	•	*n/a*	*n/a*	10,756	312.5	40.0	3.7
Henzen *et al.* [18]	Virtex-5	xc5vlx220	○ / • / ○	•	*n/a*	*n/a*	14,799	233.0	119.3	8.1
Wang *et al.* [5]	Virtex-5	xc5vlx85t	○ / • / ○	○	32,410	2,612	10,943[2]	240.3	123.1	11.2[2]

[1] number of pipeline stages
[2] estimation based on LUTs per slice

TABLE I
COMPARISON OF NEW MODULES VERSUS RELATED PREVIOUS IMPLEMENTATIONS

of a pipelined design. Once fully initialized, this design outputs the MAC of a 128-bit AAD and a 128-bit message M every clock cycle. Feedback is not needed for 128-bit and 256-bit messages since message authentication is performed every clock cycle. To authenticate messages longer than 256 bits, the pipelined design can be adapted by 1) providing a feedback path from the last stage to the input of the first stage, 2) loading the corresponding output register with the AAD, if required and 3) scheduling the data evaluation. For this last step, input IN can receive a 128-bit message block, a 128-bit LEN value or a 128-bit PAD value.

IV. RESULTS

Both the basic GHASH module (Figure 2) and pipelined chains of modules were written in Verilog and targeted to several contemporary Xilinx Spartan and Virtex families using the standard Xilinx Synthesis Technology (XST) and ISE 13.1 flows. ModelSim 6.6a was used for design simulation both before and after FPGA place-and-route. Simulation was performed under nominal conditions of voltage (0.95V), temperature (85°C) and effort settings for mapping and place-and-route. The timing of design I/Os was omitted so modules could be considered as standalone functions.

A. Performance evaluation

Table I provides a summary of the frequency, throughput, and resource usage for the new designs versus previously-published results. Table I is provided as a high-level reference since the listed previous designs are the most similar in nature to the ones reported in this work. Unlike the previous efforts, our new approach is optimized both for FPGA LUTs and specialized on a per-key basis. Because there is a large diversity of previous implementations, we compare our design to GF(2^{128}) multiplier-only designs (M), full GHASH architectures (G) and key-dependent structure (K). Note the filled dots in the M / G / K column. Additionally, some designs reported performance results prior to place-and-route, as indicated in the table (PAR column). In some cases, only LUTs are reported in the previous work. Slice counts are estimated in these cases by considering the number of LUTs per slice in the target architecture.

To evaluate the area efficiency of our approach in the context of resource usage, we use the metric *throughput per slice* for comparison between the architectures. This metric is widely used in the cryptographic community. If increased throughput is needed for our architecture, multiple pipelined copies of our new modules could be instantiated, as discussed in Section III. For highest performance, the basic GHASH module operates at 384.6 MHz on a Virtex-6 device. The design uses 143 FFs, which includes a 128-bit output register and duplicated registers to improve fan-out, and 1,714 LUTs, which are primarily used for the GF(2^{128}) multiplier. Two processing cycles are required per output so $384.6 \times 10^6 \times 128/2$=24.6 Gbit/s of throughput is achieved.

An 8-stage pipelined GHASH module operates at 285.7 MHz on a Virtex-6. In total, it consumes 1,358 FFs and 15,414 LUTs. This implementation produces the 128-bit MAC of a 1 KByte message every clock cycle. One cycle processing is needed, so $285.7 \times 10^6 \times 128 \times 8$=292.6 Gbit/s of throughput is achieved. Overall throughput per slice is 73.4 Mbit/slice.

The HDL source code, simulation testbenches, and software tools to reproduce reported results for the two module variants are publicly available at this URL: http://code.google.com/p/ghash/

B. Comparison with Prior Work

Huo *et al.* [7] proposed a bit-parallel implementation of the GF(2^{128}) multiplier based on Arash Reyhani-Masoleh algorithm. Wang *et al.* [5] reported a high-speed GHASH architecture based on an advanced multiplication algorithm. The Mastrovito multiplier used 32,410 LUTs to provide a throughput of 123.1 Gbit/s (11.2 Mbit/slice). These two papers report performance results prior to place-and-route, requiring throughput estimation. Zhou *et al.* [16] described a Virtex-5 GF(2^{128}) multiplier implementation using the Karatsuba-Ofman algorithm with 4,628 slices and a throughput of 41.5 Gbit/s (8.9 Mbit/slice). Chen *et al.* [17] proposed an 4-stage pipelined GHASH implementation on a Virtex-4 with 10,756 slices and an authentication throughput of 40 Gbit/s (3.7 Mbit/slice). Henzen and Fichtner [18] proposed a complete parallel-pipelined AES-GCM module for 100 Gbit/s Ethernet applications. The GHASH function is based on a composite field, consumes 14,799 slices and gives a 119.3 Gbit/s throughput (8.1 Mbit/slice) on a Virtex-5.

For all measured comparisons, our new GHASH module shows improved throughput per area. Raw throughput on a Virtex-5 of 238.1 Gbit/s is superior to the highest reported previous approach at a reduced GHASH area. The specialized nature of our design facilitates implementation on low cost Spartan-3 devices. The pipelined version of the module achieves about 14 Gbit/s of throughput and uses 4,875 LUTs packed in 2,484 slices.

While our results are interesting in terms of both reduced overhead and high throughput, the effectiveness of our design is tied to the frequency in which the hash key H, derived from secret key, K, changes. From AES-GCM specifications, a single key allows for the authentication of 4 GByte/s (32 Gbit/s) of data over 64 years without compromised security [2]. This term is generally much longer than a typical digital system lifetime. However, as outlined in Section II-D, key changes may be required more frequently depending on the target application. The reconfigurable nature of the FPGA makes changes to hash keys which have been embedded in hardware possible. A new bitstream can be dynamically loaded into the FPGA, when needed. Other key customization approaches [5], [7], [15], [16], [18] require only a key register reload.

V. Conclusion

In this paper, a new FPGA-optimized implementation of a GHASH message authentication block has been presented. The module takes advantage of the specialization offered by FPGAs to customize GHASH hardware based on a secret key. Significant improvements in throughput per area versus previous approaches are achieved for a variety of Xilinx architectures.

References

[1] Intel, "Thunderbolt™ Technology," 2011.

[2] D. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM). Updated submission to NIST, Modes of Operation Process," 2005.

[3] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter mode (GCM) and GMAC," *NIST Special Publication 800D-38D*, Nov. 2007.

[4] M. Wegman and L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 265–279, Jun. 1981.

[5] J. Wang, G. Shou, Y. Hu, and Z. Guo, "High-speed architectures for GHASH based on efficient bit-parallel multipliers," in *Proceedings: IEEE International Conference on Wireless Communications, Networking and Information Security*, Jun. 2010, pp. 582–586.

[6] C. Paar, "Implementation options for finite field arithmetic for elliptic curve cryptosystems," in *Proceedings: Elliptic Curve Cryptosystems Workshop*, Nov. 1999.

[7] J. Huo, G. Shou, Y. Hu, and Z. Guo, "The design and FPGA implementation of GF(2^{128}) multiplier for GHASH," in *Proceedings: International Conference on Network Security, Wireless Communications and Trusted Computing*, Jun. 2009, pp. 554–557.

[8] V. Shoup, "On fast and provably secure message authentication based on universal hashing," in *Proceedings: Advances in Cryptology*, Aug. 1996, pp. 313–328.

[9] *Cisco ASA 5500 Series Adaptive Security Appliances Models Comparison*, Cisco Corporation, 2011. [Online]. Available: http://www.cisco.com/en/US/products/ps6120/prod_models_comparison.html

[10] R. Vaslin, G. Gogniat, J.-P. Diguet, R. Tessier, D. Unnikrishnan, and K. Gaj, "Memory security management for reconfigurable embedded systems," in *Proceedings: International Conference on Field-Programmable Technology*, Dec. 2008, pp. 153–160.

[11] K.-S. Han, K.-O. Kim, T. W. Yoo, and Y. Kwon, "The design and implementation of MAC security in EPON," in *Proceedings: International Conference on Advanced Communication Technology*, May 2006, pp. 1676–1680.

[12] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, January 1998.

[13] G. Marsaglia, "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness," 1995.

[14] P. L'Ecuyer and R. Simard, "TestU01: A C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, pp. 22:1–22:40, August 2007.

[15] Y. Lu, G. Shou, Y. Hu, and Z. Guo, "The research and efficient FPGA implementation of Ghash core for GMAC," in *Proceedings: International Conference on E-Business and Information System Security*, May 2009, pp. 1–5.

[16] G. Zhou, H. Michalik, and L. Hinsenkamp, "Improving throughput of AES-GCM with pipelined Karatsuba multipliers on FPGAs," in *Proceedings: International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, Mar. 2009, pp. 193–203.

[17] T. Chen, W. Huo, and Z. Liu, "Design and efficient FPGA implementation of Ghash core for AES-GCM," in *Proceedings: International Conference on Computational Intelligence and Software Engineering*, Dec. 2010, pp. 1–4.

[18] L. Henzen and W. Fichtner, "FPGA parallel-pipelined AES-GCM core for 100G Ethernet applications," in *Proceedings: European Solid State Circuits Conference*, Sep. 2010, pp. 202–205.