Small Lightweight Hash Functions in FPGA

Carlos Andres Lara-Nino, Miguel Morales-Sandoval, and Arturo Diaz-Perez
CINVESTAV Campus Tamaulipas. Victoria, Tamaulipas. México.
Email: {clara, mmorales, adiaz}@tamps.cinvestav.mx

***Abstract*—This paper presents hardware realizations of two lightweight hash function families on FPGA:** SPONGENT **and LHash. The assessment provided for both cryptographic primitives is in terms of area, performance, and energy consumption, when implemented in LUT-4 and LUT-6 FPGA technology for equivalent security levels. To the best of our knowledge, this paper reports the most compact** SPONGENT **FPGA implementation and the first FPGA implementation of LHash.**

***Index Terms*—Lightweight Hash Function, Low-area, Low-energy, FPGA.**

I. Introduction

Security is a critical issue in the envisioned applications of the Internet of Things (IoT) such as in sensing, stock monitoring, mobile health, and military applications, where sensitive information is often being processed or transmitted by the underlying constrained devices.

Traditionally, cryptography provides the means to secure sensitive data. However, the resources required by cryptographic algorithms to operate often exceed the capabilities of IoT devices in regards to area and power. Hence, lightweight cryptography has been proposed as an alternative [1]–[3].

Hardware realizations of cryptographic algorithms are intended to offload computations from the main processor, being ASIC and FPGA the main alternatives. Although ASIC implementations are faster than those based on FPGAs, the latter are sometimes preferred over ASIC for cryptographic applications due to their inherent properties of reconfigurability, short time to market and in-house security [4].

In this paper we tackle the implementation of lightweight hash functions on FPGA. These primitives have been utilized to provide integrity and authentication security services.

We selected the hash function SPONGENT, which has the smallest area recorded in ASIC [5]. We also consider in this study LHash, which is a high-performance and small cryptographic hash function. Although the implementation of hash functions in FPGA has been addressed before [6], the difference in our work is that we seek a broader, multidimensional implementation assessment that takes into consideration performance and energy consumption analysis. We consider this requirement crucial as our proposed hash modules are intended to operate in resource constrained environments (i.e., the IoT).

The rest of the paper is structured as follows. Section II presents details about the hash algorithms under study. Section III describes the proposed hardware architectures. Section IV describes our evaluation methodology and presents our findings. Section V concludes the paper.

II. Hash functions

A hash function is a function that compresses data. It takes an almost arbitrary-length message as input and produces a digest of fixed-length. A cryptographic hash function features resistance to collision, pre-image and 2nd pre-image attacks [7]. Secure hash functions generally impose high overhead on the underlying system.

A. SPONGENT

SPONGENT is a family of lightweight hash functions with hash size n of 88, 128, 160, 224, and 256 bits based on a sponge construction [8] instantiated with a PRESENT-type permutation [5].

We denominate STATE to the internal data block undergoing processing. The input message blocks are XOR-ed with the r rightmost bits of the STATE. The same r bits are used to generate the hash output. The b bits in the STATE are processed with an R rounds internal transformation. The SPONGENT family has five parameters (n, b, c, r, R) to instantiate a specific hash function, these are shown in Table I.

SPONGENT-88 has pre-image resistance of 2^{80}, second pre-image resistance of 2^{40}, and collision resistance of 2^{40}. Even though these security levels can be considered low for cryptography standards, they can be of utility for constrained applications. This is based on the assumption that certain applications would chose to forfeit some security in the pursuit of efficiency.

SPONGENT follows three steps: initialization, absorbing, and squeezing. Its internal transformation SPONGENT-$f[r + c]$, denominated $\pi_b : \mathbb{F}_2^b \rightarrow \mathbb{F}_2^b$, is a R-round transformation of a b-bit data block, as presented in [5].

The internal transformation π_{88} consists of the functions $\texttt{sBoxLayer}_b$ and $\texttt{pLayer}_b$ and the finite addition of a Linear-Feedback Shift Register (LFSR) value $\texttt{lCounter}_b(i)$ to the state.

B. LHash

LHash is based on the sponge construction instantiated with a Feistel round transformation. It is a family of lightweight hash functions with hash size n of 80, 96, and 128 bits.

If r is the length of the input message blocks and c is the size of the capacity, then $b = r + c$ is the size of the fixed transformation LHash-$f[r + c]$ and r' is the output size for each output digest block. The different parameterizations of LHash are defined by four characteristics (n, b, r, r'). These parameters are presented in Table I.

978-1-5386-2311-4/18/$31.00

TABLE I
DIFFERENT CONFIGURATIONS OF SPONGENT AND LHASH. ASIC IMPLEMENTATION RESULTS FROM RELATED WORKS.

Hash function	Ref.	n (bits)	b (bits)	c (bits)	r (bits)	r' (bits)	R	Security(bit)			Area (GE)	Thr. (kbps)
								PRE	2nd PRE	COL		
SPONGENT-88	[9]	88	88	80	8	8	45	80	40	40	738	0.81
LHash-96	[10]	96	96	80	16	16	18	80	40	40	817	2.40

LHash-96 has pre-image resistance of 2^{80}, second pre-image resistance of 2^{40}, and collision resistance of 2^{40}, which matches the security levels of SPONGENT-88 thus enabling a fairer comparison.

LHash follows three steps: initialization, absorbing, and squeezing. LHash-$f[r+c]$ stands for a fixed internal permutation F_b (b=96 or 128). It is constructed using an 18-round Feistel structure. The round transformation is detailed in [10].

The round function of LHash represents a permutation where the b-bit input is first split into two halves $X_1||X_0$. Then, for $i = 2, 3, \ldots, 19$, it follows $X_i = G_b(P_b(X_{i-1} \oplus C_{i-1})) \oplus X_{i-2}$. At the end, $X_{19}||X_{18}$ is the output of the permutation.

III. ARCHITECTURAL DESIGN

This section describes the hardware architectures proposed in this paper to implement the selected hash functions.

A. SPONGENT

The architecture developed for SPONGENT-88 is based on a hardware loop with a single register. In the initial phase the STATE value is the all zeros word. At the first round the input block is XOR-ed with the value in the register. At each round the register is then XOR-ed with the contents of an LFSR and processed by the substitution and permutation layers. Figure 1 illustrates this design, note that the LFSR is shown as $\texttt{lC}_{88}$.

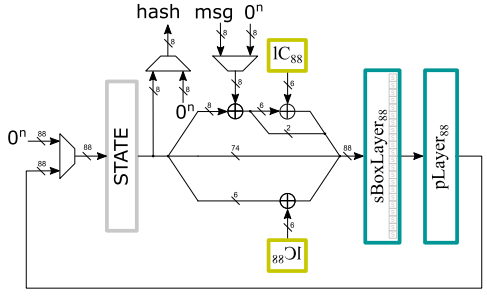

Fig. 1. Hardware architecture for SPONGENT-88.

The LFSR counter is generated by a 6-bit register clocked each active round. Its value is XOR-ed with the least significant bits of the STATE and its reversed value is XOR-ed with the STATE's most significant bits. Figure 1 shows two LFSR, however, that is only for illustrative purposes since in practice a single one is used.

The substitution layer is formed by 22 4-bit substitution boxes which process the STATE in parallel. A common strategy for minimizing area is to reduce the number of substitution boxes, but this has a linear increase in the latency. A round of SPONGENT-88 requires 45 cycles. By reducing the number of substitution boxes to half, the latency would be doubled. Since this trade-off affects the latency and energy consumption considerably, that strategy was not implemented.

The permutation layer is a simple wiring which can be straightforwardly implemented with little cost.

The output is taken directly from the output of the register. To reduce the switching activity at the output, we opted to include a mask, thus improving the energy consumption at the cost of a few additional hardware resources.

The latency calculation for this architecture is presented in Table II. From left to right, the columns in the Table represent: the architecture, the cycle count required to input a 256-bit message, the cycle count required to input the padding, the cycle count required to produce the hash output of length n (88 or 96 bits), and the total latency.

TABLE II
LATENCY CYCLES FOR DIGEST GENERATION

Architecture	Absorb 256-bit	Absorb padding	Squeeze n-bits	Total
SPONGENT-88	1440	45	495	1980 cycles
LHash-96	288	18	108	414 cycles

B. LHash

The Feistel structure utilized in LHash is more complex than the one in SPONGENT. We decided to divide the main register in two smaller ones to represent the partitioning of data in the internal round. The input message is always XOR-ed with the lower part of one of these registers, and the output is generated from the same position. At each round, the value of one of the two registers is XOR-ed with a round constant and processed by the permutation, substitution, and Maximum Distance Separable (MDS) layers. This result is then XOR-ed with the original contents in the register and returned to the same register. Figure 2 illustrates this architecture.

The round constant is formed from a 5-bit LFSR expanded to 16-bit. These 16 bits are only wired and not stored to reduce resource usage. The round constant is XOR-ed with the most significant bits of the active STATE.

The permutation layer is formed by four 12-bit permutations applied in parallel to the 48-bit STATE. This is also a wiring.

The substitution layer contains 12 4-bit substitution boxes to process the active STATE in parallel.

The number of substitution boxes required in the LHash implementation can be reduced, i.e. halved, at the cost of an increased latency. Since LHash-96 requires fewer cycles per round than SPONGENT-88 the trade-off might be considered

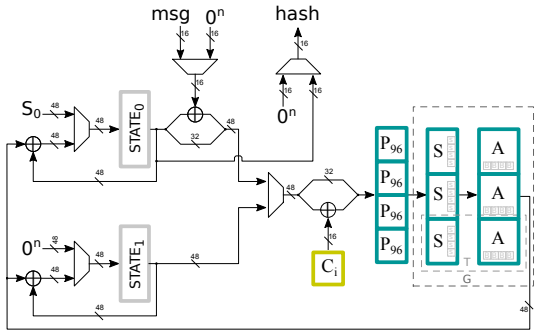

Fig. 2. Hardware architecture for LHash-96.

adequate. We believe that to achieve optimal results it would be required to streamline the design and use a datapath with constant width. This means reducing the permutation and MDS layers width as well. As previously pointed out, the permutation layer is made of four smaller permutations thus it can be reduced if required. The MDS layer, however, is made of three 16-bit transformations. This means it cannot be reduced to the same width of the substitution and permutation layers. Since it is not possible to obtain a unified datapath with an optimal number of cycles per round for LHash-96, these area optimizations were discarded. The reader should note that this optimization is achievable for other instances of LHash.

The MDS layer represents the application of three 16-bit A functions in parallel over the STATE. These functions are mostly wires and XOR gates plus six 4-bit field multipliers. Since one of the multiplicands is a constant, the multiplication/reduction can also be wired. Each 16-bit function requires 11 1-bit XOR operations. In total, 33 XOR gates are utilized in this layer.

The output is taken directly from one of the registers. In the same fashion as it was made for the SPONGENT architecture, a mask is added at the output to mitigate the switching activity.

Table II presents the latency details for this architecture.

IV. Experimental evaluation

A. Implementation strategies

The proposed designs were described using VHDL. Reduced design complexity was the main strategy to achieve small implementation sizes. The main elements in each design were constructed as modules, interconnected with a minimal number of multiplexers, and controlled by wired FSMs.

An FSM was used as main controller for each hash architecture. This unit generated the enable signals for routing the data through the datapath. Since in sponge constructions the same internal transformation is applied to process the input and to produce the output, another FSM was used as round controller. The round controller required a small counter to determine the number of iterations applied over the state.

External signals were used to indicate the desired input and output lengths. This allowed both the flexibility in the incorporation of the hash module to different architectures and the reduction of the control complexity.

B. Experimental setup

a) Environment: The architectures were synthesized for Xilinx FPGAs using the ISE Design Suite 14.3. As computing platform, two low cost development boards were considered. The Spartan-3 (xc3s50) was used to evaluate the architectures in LUT-4 FPGAs. For LUT-6 FPGAs, the Spartan-6 (xc6slx16) was considered.

b) Synthesis criteria: The synthesis process was configured with *Area* as *Optimization Goal* and *High* as *Optimization Effort* for all the implementations. The use of FPGA embedded RAM/ROM resources was disabled to achieve a fairer comparison with the state of the art.

c) Configurations: Two designs were studied: SPONGENT-88 and LHash-96. Each one includes I/O mechanisms that allow using the hardware module as an independent core. Each implementation was made targeting a specific FPGA board to generate the physical constraints file. The source files for each implementation are available upon request to the lead author.

d) Metrics: The metrics of interest are the implementation size, performance and energy consumption. *Slices* (SLC) are used as area units, but the results in LUT and Flip-Flop (FF) usage are also presented. Performance is expressed in terms of *latency* and *throughput*. For energy consumption, we provide the calculation of *total power* and the *energy* required to produce a digest. Two derived metrics are used: the *throughput-per-slice* is a relation of the implementation size and the performance; the *energy-per-bit* is the estimation of energy spent to process the input message.

In this work, we consider a frequency of 100 KHz to compute performance and energy consumption, which is a value commonly used in the literature [11].

The maximum throughput (Thr) is a function of the maximum frequency (FMAX), the latency in cycles (LAT), and the message size (BSIZE). It is computed as $\text{Thr} = \frac{\text{FMAX} \times \text{BSIZE}}{\text{LAT}}$. The throughput-per-slice (Thr/SLC) is calculated as $\text{Thr/SLC} = \frac{\text{Thr}}{\text{SLC}}$.

The tool Xilinx XPower Analyzer 14.3 was utilized to estimate the static, dynamic, and total power (POW) of each implementation for a frequency of 100 KHz. The software reported *high* overall confidence levels for all the estimations.

The energy (ENE) spent by an implementation to process a message, considering an operational frequency of 100KHz, is calculated as $\text{ENE} = \frac{\text{POW} \times \text{LAT}}{100\ \text{KHz}}$. The energy-per-bit is computed as $\text{ENE/bit} = \frac{\text{ENE}}{\text{BSIZE}}$.

C. Implementation results

The implementation results for the architectures under evaluation are shown in Table III, which also shows a comparison with the state of the art in FPGA implementations of SPONGENT with the same security level. Table IV features implementation results for power and energy consumption.

The implementation of SPONGENT-88 achieved smaller FF, LUT, and SLC count than LHash for the two FPGAs utilized.

In regards to throughput, for a constant frequency of 100KHz, LHash-96 outperformed SPONGENT by approxi-

TABLE III
RESOURCE USAGE AND PERFORMANCE FOR THE HASH FUNCTIONS UNDER EVALUATION. RESULTS OBTAINED FROM PLACE-AND-ROUTE.

Work	Design	Message (bits)	Digest (bits)	r	r'	Internal rounds	FF	LUT	SLC	FMAX (MHz)	LAT (cycles)	Thr (Mbps)	Thr* (Kbps)	Thr*/SLC Kbps/Slice
xc3s50-5cp132														
[11]	SPONGENT-88	256	88	8	8	45	-	-	116	90	1980+	11.60	12.93	0.11
This work.	SPONGENT-88	256	88	8	8	45	**104**	**143**	**74**	**227**	1980	29.32	12.93	0.17
This work.	LHash-96	256	96	16	16	18	110	380	203	97	**414**	**60.12**	**61.84**	**0.30**
xc6slx16-3csg324														
[6]	SPONGENT-88	256	88	8	8	45	-	-	26	**309**	1980+	39.95	12.93	0.50
This work.	SPONGENT-88	256	88	8	8	45	**104**	**71**	**20**	302	1980	39.03	12.93	0.65
This work.	LHash-96	256	96	16	16	18	110	234	67	142	**414**	**88.09**	**61.84**	**0.92**

* Using a frequency of 100KHz.
\+ An optimal latency is considered since the data is not publicly available.

TABLE IV
POWER AND ENERGY CONSUMPTION FOR THE HASH FUNCTIONS UNDER EVALUATION. RESULTS OBTAINED FROM PLACE-AND-ROUTE.

Work	Design	Message (bits)	Digest (bits)	r	r'	Internal rounds	LAT (cycles)	POW* (mW)			ENE* (uJ)	ENE*/bit (uJ/bit)
								Static	Dynamic	Total		
xc3s50-5cp132												
This work.	SPONGENT-88	256	88	8	8	45	1980	**27.25**	**0.81**	**28.06**	555.59	2.17
This work.	LHash-96	256	96	16	16	18	**414**	**27.25**	1.64	28.89	**119.60**	**0.47**
xc6slx16-3csg324												
This work.	SPONGENT-88	256	88	8	8	45	1980	**19.91**	**1.28**	**21.19**	419.56	1.64
This work.	LHash-96	256	96	16	16	18	**414**	**19.91**	2.09	22.00	**91.08**	**0.36**

* Using a frequency of 100KHz.

mately 5 times. In the efficiency metric of Thr*/SLC for the Spartan-3, LHash-96 almost doubled the efficiency of SPONGENT. For the Thr*/SLC in the Spartan-6 FPGA, LHash-96 obtained an advantage of 30% over SPONGENT-88.

However, for power estimation, SPONGENT-88 obtained the smallest expenditures for both FPGAs utilized. When the latency was factored for the energy consumption, the advantage turned in favor of LHash-96. Since the message size is constant, the efficiency in ENE*/bit also showed better results for LHash-96.

D. Results comparison

In the literature we found two implementations of SPONGENT-88 in FPGA [6], [11]. No implementations of this type were found for LHash. These works provide brief results in area and performance and are used as reference in this section. Since we do not have access to the implementation files of aforementioned works, we use here the published information. The Xilinx toolchain version utilized in [6], [11] is not reported in the sources.

Compared to [11] and [6], our SPONGENT-88 implementation achieved smaller resource usage in SLC and higher efficiency in throughput-per-slice. The main difference in our work compared to that reported in [11] is the architectural design, which in our case has a control with reduced complexity. Our proposal appears to be similar to the architecture described in [6], following similar implementation processes using the Xilinx tool chain and a Spartan-6 FPGA. However, our design requires 23% less slices.

In the case of the Spartan-3 board, our implementation improved the results in [11] by 42 SLC (36%) and requires 6 SLC (23%) less than [6] for the Spartan-6 FPGA.

V. CONCLUSION

In this paper we evaluated the hardware design of two lightweight hash functions, well suited to be used as building blocks of security schemes in the envisioned IoT applications.

The designs were crafted considering adequate trade-offs between resource usage, latency, and energy consumption. Both architectures were provided with data and control ports which allows them to be used in larger systems as a hash core.

Less cycles per round and higher I/O rates were the key aspects for obtaining the high performance and low energy consumption of LHash-96. As a result, this hash function came in top of the two efficiency metrics proposed. As far as we know, our SPONGENT-88 design implemented in the Spartan-6 FPGA is the smallest ever reported for any hash function.

REFERENCES

[1] T. Eisenbarth, C. Paar, A. Poschmann, S. Kumar, and L. Uhsadel, "A Survey of Lightweight Cryptography Implementations," *IEEE Des. Test*, vol. 24, no. 6, pp. 522–533, Nov. 2007.

[2] D. Maimut and K. Ouafi, "Lightweight Cryptography for RFID Tags," *IEEE Security & Privacy*, vol. 10, no. 2, pp. 76–79, Mar 2012.

[3] C. Alippi, A. Bogdanov, and F. Regazzoni, "Lightweight Cryptography for Constrained Devices," in *ISIC 2014*, 2014, pp. 144–147.

[4] D. B. Roy, P. Das, and D. Mukhopadhyay, "ECC on Your Fingertips: A Single Instruction Approach for Lightweight ECC Design in GF(p)," in *SAC 2015*, 2015, pp. 161–177.

[5] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *CHES 2007*, 2007.

[6] B. Jungk, L. R. Lima, and M. Hiller, "A systematic study of lightweight hash functions on FPGAs," in *ReConFig'14*, Dec 2014, pp. 1–6.

[7] M. Bellare and P. Rogaway, *Introduction to Modern Cryptography*, 1st ed., 2005.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Sponge Functions," Ecrypt Hash Workshop 2007, 2007.

[9] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, "SPONGENT: A Lightweight Hash Function," in *CHES 2011*, 2011, pp. 312–325.

[10] W. Wu, S. Wu, L. Zhang, J. Zou, and L. Dong, "LHash: A Lightweight Hash Function," in *Inscrypt 2013*, 2014, pp. 291–308.

[11] M. Adas, "On the FPGA-Based Implementation of SPONGENT," [Online] http://ece.gmu.edu/coursewebpages/ECE/ECE646/F11/project/F11_presentations/Marwan.pdf, 2011, [Last access] 06-26-2017.