Power, Performance and Area Analysis of Sponge Based Lightweight HASH Function

Heera G. Wali* and Nalini C. Iyer
School of Electronics and Communication, KLE Technological University, Hubli, Karnataka, India
Email: heerawali@kletech.ac.in* (H.G.W.); nalinic@kletech.ac.in (N.C.I.)

***Abstract*—The Internet of Things refers to uniquely distinguishable devices communicating with each other. Communication is largely on resource-constraint devices which would generate large volumes of data, the security, authenticity, and integrity of devices being essential. SPONGENT-88 variant of the SPONGENT family is a lightweight Hash algorithm suitable for authenticating resource-constrained devices in an IoT environment or WSN (Wireless Sensor Network) as it has the smallest footprint in comparison with the sponge-based Hash algorithm. Conventionally most of the security primitives are realized using more sequential logic which leads to an increase in the area occupied and latency. This work proposes a serialized architecture with a balance in sequential and combinational logic that uses lesser flipflops with a 21% increased performance for throughput/slice in comparison with reported works on target LUT-6 technology on FPGA, realized on the 14.2 ISE design suite. The work has a 4.6% decrease in quiescent power and dynamic power reported on the 14.2 ISE design suite for LUT-6 technology. This work has also achieved a Gate-Equivalence (GE) utilization for the proposed design on TSMC180 technology for a data path of 88-bit, while the proposed work reports 1204GE. In this work, a scaled-up architecture of the proposed SPONGENT-88 is realized for SPONGENT-128 and SPONGENT-224 as these variants provide improved security bound and is in comparison with the security provided by SHA-1 and SHA-2 algorithms. The scaled architecture for the latter variants of SPONGENT algorithms has achieved reduced Gate-Equivalence (GE) in comparison with the reported work.**

***Index Terms*—Hash algorithm, lightweight authentication, sponge-based authentication**

I. Introduction

Lightweight Cryptography (LWC) is seen to be spanning into existence as the last two decades saw the proliferation of a variety of interconnected devices in an environment known as the IoT(Internet-of-Things)/ WSN(Wireless Sensor Networks). With these devices humongous amount of data is been generated. However, due to a lack of effective security and privacy protocols, most IoT devices are prone to hacking. Therefore addressing the security and privacy needs of these connected low-cost pervasive devices with limited computations is of prime importance. The conventional cryptographic primitives that were designed were computationally complex and hence were not applicable to the security and privacy needs in the IoT/WSN paradigm.

Lightweight cryptography is not bound to replace conventional cryptography while they are efficient but was designed to address the specific use cases at the cost of having a narrower range of applicability. The goal of lightweight cryptography is to enable a wide range of connected devices with different applications such as health monitoring systems, automated management of supply chains, public transportation, telephone cards, etc.

The connected edge devices in IoT/WSN are possible to identify because of different tagging technologies, making all the end nodes/physical objects connect to communicate and share information. The communication between the end devices must be secured in IoT/WSN with confidentiality, integrity, and authentication services. Several new factors are constrained for operating under these environments like limited computational power, RAM size, and ROM size with lesser footprint. But these constrained IoT devices/end nodes use traditional security measures [1], which computationally require more power and memory. Hence IoT-enabled devices were the medium for the emergence of a new field, lightweight cryptography(LWC). With this, a few software and hardware implementations of lightweight ciphers are designed for IoT applications. These are broadly classified as hash functions, stream ciphers, and block ciphers. Software implementations have lower costs and provide more flexibility in manufacturing and maintenance. However, the literature proposes two core categories of lightweight cryptography depending on different types of applications with hardware (HW) and software (SW) attributes.

- Ultra LWC: For a specific application and security requirement in a highly resource-constrained environment (e.g. specific microcontroller, for only one of the security measures Hash/encryption/ authentications) occupying a footprint lesser than 1000GE for security primitives.
- Ubiquitous LWC: IoT end nodes (e.g. FPGA, general-purpose microcontroller, one/more security primitives) occupying a footprint of less than 3000GE for security primitives at the end nodes [2].

The Ultra LWC finds applications in domains like IIoT (Industrial IoT) where the security and privacy of the connected end nodes need to find be very specific and

Manuscript received December 2, 2022; revised February 3, 2023; accepted February 12, 2023.
*Corresponding author: Heera G. Wali

highly resource constrained. Ubiquitous LWC is directed towards IoT/WSN end nodes, that have varying end application that requires different security measures. Most of the research is primarily based on ubiquitous lightweight cryptography. Although ultra-lightweight cryptography is implementable with existing regular resources. There is always a need to provide a tailor-made/optimized solution for these lightweight cryptographic primitives to have a lower footprint and improved performance.

Lightweight cryptography has two forms, likewise to conventional cryptography, known as symmetric and asymmetric. A secure hash algorithm is one part of cryptography protocols that require quick processing and authentic data transmission. A lightweight cryptography Hash function is required in IoT applications that involve intensive and sensitive data transactions, as these devices have to primarily authenticate themselves to communicate. This work focuses on lightweight Hash primitives for IoT/WSN based applications.

One of the areas where lightweight cryptography finds its applicability is IoT-based healthcare is an area where lightweight cryptographic primitives can be harnessed. Here a large amount of private data is being collected, and the security of the healthcare services provided can impact the well-being of the patient's life [3]. These devices need different cryptographic primitives such as encryption, authentication, and authenticated encryption to be implemented and it is challenging, because of resource constraints as it also connects the edge/fog computing layers to medical databases and cloud computing servers.

Blockchain is another emerging technology that is being deployed in the IoT architecture which has yielded numerous issues, challenges, and advantages [4]. Authentication of the connected nodes is essential in integrating blockchain in IoT. Sponged-based Hash primitives are found to be more suitable in terms of performance and energy consumption for blockchain based applications which also facilitates building a robust IoT architecture [5].

The paper is organized as follows, the Introduction section followed by related works of the lightweight Hash algorithm in Section II, in Section III background of the sponge-based Hash construction, the proposed architecture for the SPONGENT-88 algorithm in Section IV, and results and discussion in Section V.

II. Related Works

Many advancements in the field of lightweight cryptography have been observed with the end goal being to tailor-make the conventional cryptographic algorithm for resource-constrained platforms/end nodes. Lightweight algorithms must be designed to have a lower footprint with improved performance. Although many lightweight cryptographic algorithms have been published the security analysis is performed at the algorithmic level and still finds a need to keep in consideration the physical attacks on these primitives [6].

S. Aruna *et al.* [7] have outlined the hardware and software implementation challenges of lightweight cryptographic primitives. They provide the classification of Hash function construction mode with lightweight Hash function for real-time applications.

The lightweight authentication algorithms can be classified under three constructions a) Davies–Meyer, b) Merkle-Damgard, and c) Sponge. The lightweight Hash algorithms under these different constructions for resource-constrained environments which occupy areas less than 3000GE are keccak-f[200] [8], ARMADILLO [9], GLUON [10], PHOTON [11], QUARK [12], LHash [13], and SPONGENT [14].

The authors of the work [15] report a comparative analysis of the sponge-based Hash algorithms for IoT applications. They present a comparison of the algorithms: keccak-f[200] [8], GLUON [10], PHOTON [11], QUARK [12], LHash [13], and SPONGENT [14] in terms of throughput, power, and area where SPONGENT-88 [14] is having lowest power consumption with 1.57 μW among the Hash algorithms with an area of 738GE.

SPONGENT [14] is also the underlying primitive used in the NIST LWC finalist algorithm Elephant [16]. It's a block cipher based on permutation and uses encrypt-then-MAC procedure for authentication.

The details of the various author's contributions to the Hash algorithms keccak-f[200] [8], ARMADILLO [9], GLUON [10], PHOTON [11], QUARK [12], LHash [13], and SPONGENT [14] which are under 3000GE are discussed below in this section.

Stephane Badel *et al.* [9] propose a general-purpose cryptographic function design for RFID (Radio Frequency Identification) and sensor networks with different design construction for applications like challenge-response protocols, hashing, and digital signatures, built on Merkle Damgard construction. A serialized design of ARMADILLO occupies 2923GE on 180nm technology node and has better performance in throughput and throughput per GE with frequency constrained at 100kHz in hashing design construction in comparison with SHA-1, and SHA-256.

Keccak-f[200] and Keccak-f[400] by Bertoni *et al.* [8] are winners of NIST SHA-3 Hash function competition. Keccak-f[200] produces a 64-bit Hash digest with an internal state permutation of 200-bit, the permutation function is based on sponge construction and finds itself a suitable Hash algorithm for a resource-constrained environment. Keccak-f is been designed for different permutation sizes and variable-length output digest respectively while keeping the permutation function the same. A serialized implementation of Keccak-f[200] is realized on 130nm technology using 2520GE for a frequency constrained at 100kHz.

PHOTON [11] (Sponge-based construction) was designed with sponge-based construction with AES-like primitive as an internal unkeyed permutation. It has been obtained as the most compact Hash function with 1120GE with 64-bit security bounds on pre-image and collision attacks with a new method for column mixing layer in a serial way, which led to lowering the area requirement as compared to the original AES algorithm.

PHOTON has been proposed with variants ranging from Hash digest size of 80-bit to 256-bit with pre-image security bounds ranging from 64-bit to 224-bit respectively. The smallest variant of PHOTON occupies 865GE and the largest occupies 4362GE.

Berger *et al.* [10] proposed a sponge-based Hash algorithm GLUON after the publishing of the PHOTON and SPONGENT algorithms. It is derived from two stream ciphers F-FCSR-v3 and X-FCSR-v2. GLUON is designed for three different message digests with varying security levels. The lightest instance is GLUON-128/8 with 64-bit security level that fits in 2071GE and the heaviest variant GLUON-224/32 performs well on the software platform. This is based on Feedback with Carry Shift Register (FCSR) while the design is heavier than the basic building blocks of Quark and PHOTON, the well-proven design of FCSRs has been highlighted as the strength of the GLUON algorithm. It is regarded as better in area than Quark variants. In terms of throughput, it's similar to the 64-bit of the Quark variant only. When compared to PHOTON the area achieved by GLUON is not comparable at all but performs better in throughput.

Authors of QUARK (Sponge-construction) Aumasson *et al.* [12] designed three variants U-QUARK, D-QUARK, and S-QUARK. Among the variants where the lightest version is U-Quark fits with 1379GE with a minimum of 64-bit security against attacks. S-Quark has higher security bound with 112 bits realized with 2296GE on 180nm technology node. When compared with the lightweight sponged designs SPONGENT and PHOTON which appeared after the publication of Quark, Quark has a better security margin compared to PHOTON and better throughput than SPONGENT but has a higher footprint than both PHOTON and SPONGENT.

LHash by Wenling *et al.* [13] is a lightweight Hash algorithm LHash that employs a Generalized Feistel structure (GFS) in the internal permutation as this structure provides good diffusion speed in the permutation layers. This Hash function supports three different digest sizes: 80-bit, 96-bit, and 128-bit, providing preimage security from 64-bit to 120-bit, and second preimage and collision security from 40-bit to 60-bit. LHash requires about 817 GE and 1028 GE with a serialized implementation. In faster implementations based on function T, LHash requires 989 GE and 1200 GE with 54 and 72 latency cycles per block, respectively. LHash is known to have the lowest energy consumption among the existing lightweight Hash algorithms with moderate security levels.

Lesamnta-LW [17] is a 256-bit Hash function designed for low-end devices with an 8-bit processor. The basic building block under Lesamnta is AES with 256-bit plaintext and a 128-bit key. Lesamnta-LW is the lightweight version of Lesamnta. Lesamnta -LW uses a smaller key size than the block size for the underlying algorithm, whereas the LW1 version does not have a feedforward of inputs hence leading to a reduction in memory. It provides 120-bit security bounds against collision and preimage attacks. It has been realized on 90nm technology node occupying 8.24K gates. Lesamnta-LW was been designed for applications on low-cost devices that require higher security bounds. Even though the gate-equivalence of Lesamnta is more than 3000GE, it's an LWC NIST [18] recommendation under lightweight Hash functions, which can be deployed in the software platform.

Hash functions can also be realized using block cipher as the underlying primitives. This is being realized in Davies-Meyer construction where a block cipher is used to construct a hash primitive. Bogdanov *et al.* [19] constructs a Hash function from a block cipher PRESENT with 64-bit and 128-bit Hash outputs. The design is been proposed with two combinations under 64-bit Hash digest as DM-PRESENT-80 and DM-PRESENT-120 using the Merkle-Damgard construction with the key size of 80-bit and 120-bit respectively. They propose H-PRESENT-128 under 128-bit Hash digest with a key size of also 128-bit. The serialized and parallel architecture of DM-PRESENT and H-PRESENT achieve a footprint of 1600GE to 4256GE under various variants of the above-mentioned Hash algorithms.

SPONGENT [14] was proposed in 5 designs (groups) with 13 variants to meet different collision and secondary preimage resistance for meeting various implementation constraints. It's known to be occupying the lesser area among lightweight sponge-based algorithm Quark and is comparable with the area of PHOTON. The group of all Spongent variants for the respective design group produces the same *n*-bit Hash digest. The SPONGENT-88 variant functions are designed with low preimage security requirements for highly restricted resource environments. They can be ideally used in RFID protocols and for PRNGs. For highly constrained applications with low and middle requirements for collision security SPONGENT-128 and SPONGENT-160 can be the best fit. The design rationale of the SPONGENT-128 and SPONGENT-160 functions also provides compatibility with the SHA-1 [8] interfaces. The parameters of SPONGENT-224 and SPONGENT-256 are relatable to those of a subset of SHA-2 [8] and SHA-3 [8], hence making Spongent functions compatible with most lightweight embedded applications. The Hash function in general has received very less attention from cryptanalysis.

SPONGENT S-box was classified as one of the potentially good S-box by Zhang *et al.* [20], as the S-box is the only nonlinear component in the cryptographic primitive and plays an important role against differential cryptanalysis. According to the state-of-art differential, cryptanalysis is more important for Hash function than linear cryptanalysis. Therefore, SPONGENT lightweight Hash algorithm has the merits of addressing the resource-constrained requirements of end nodes/WSN for different applications. This work focuses to achieve improved throughput and lower footprint in comparison with the state-of-art work of the SPONGENT-88 algorithm. The work proposes a serialized architecture for the SPONGENT-88 variant of the SPONGENT family for having improved throughput and footprint under ultralightweight cryptography. The same architecture is

being scaled for different SPONGENT designs and it achieves an improved footprint in comparison with the original work [14].

III. Background

It's more difficult to implement a Hash function than a block cipher because the internal state size of the Hash function is much larger and requires a greater number of flip-flops/registers for realizing the same. For example, the SHA-3[8] uses a 1600-bit internal state whereas any lightweight block cipher would use a 64-bit block size. But when it comes to the security provided by Hash-based algorithms in terms of pre-image, secondary pre-image, and collision attacks are much more reliable than compared the lightweight block cipher.

The security measures for Hash algorithm are as follows:

- *Preimage resistant*: If H is a Hash function and x is the block of data to the function, it should be preimage resistant or one-wayness provided an output value $H(x)$ Hash value, it should be difficult to compute x to from $H(x)$. To find an input message that would map to this output. With n input-output size of the Hash function, the complexity of the attack is 2^n.
- *Secondary preimage resistance:* Also known as weak collision resistance: for a known Hash value $H(x)$ of a corresponding message x, it should be difficult to find another message $H(x')$ with the same Hash value i.e., $H(x) \neq H(x')$. Once again the complexity of the secondary preimage attack is 2^n.
- *Collision resistant:* It should be infeasible to find two messages which would lead to the same digest. This phenomenon is best illustrated with the birthday paradox problem which refers to given a set of m persons at least two of them will have their birthday on the same day, thus finding the complexity of such a collision is approximated to $2^{n/2}$.

With the growing number of connected systems, the secure communications of these systems become of paramount importance. As these devices are evolving in numbers, and hence they have a low time to market for having improved performance at the entry-level. Hence, to make these devices reliable and robust to work in different environments for several applications, it becomes necessary to provide security solutions for them [21]. A medium security level or tailor-made security is often sufficient in this area with better performance. Hence, lightweight cryptography finds its applicability here, because the algorithms are designed to have a lesser footprint with reduced security properties. A tradeoff between area and speed is achieved in the lieu of reduced security for devices under strict resource constraints such as RFID tags [22]. Lightweight Hash functions typically are used as lightweight signature schemes, RFID security protocols, or random number generators for authentication.

A. Sponge-Based Construction

A commonly used theoretical model for generating a message digest in cryptography is the well-known random oracle model. The random oracle function inputs a variable-length input message and produces a message digest of arbitrary length. Ideally, a sponge function uses a variable round-based random permutation which works on the same principle of random oracle. As shown in Fig. 1, which gives the basic sponge construction, a sponge function consists of three basic components: The initialization phase, absorption phase, and squeeze phase. The initialization phase comprises a padding function to resist length extension attacks. The absorption and squeeze phases use the permute function of variable rounds. First, in the initialization phase, the input message m is padded, which results in the padded message m of length $|l|$. Then the m is divided into $|l|/r$ blocks of m_i, each of r bits. The permute function π_b has an internal STATE b, these m_i blocks are added to the internal state, followed by the computation of the permute function. After all, blocks are absorbed, one or more blocks with rate r are derived from STATE in the squeezing phase as h_1, h_2, h_3, etc. respectively. The most interesting feature of lightweight Hash functions is its adaptable rate r, which is independent of the security level, even if the state size b, $b >= r + C$, where C is the capacity, increases (or decreases) accordingly. This parameter can be used as a design criterion for having a reduced data path of the algorithm. It also controls the number of bits that get absorbed per computation of the underlying permute function. Hence, it is possible to construct a hash function that consumes less memory compared to other well-known domain extenders such as the wide-pipe Merkle-Damgard construction, which usually requires twice the state size for the desired security level. Among the lightweight sponge-based hash functions, Quark [12] was designed with minimized state size, using the wide-pipe Merkle-Damgard construction. This usually requires twice the state size for the desired security level. Due to many inputs for the Boolean function in the round transform, it cannot be promised to achieve the smallest logic size. But the Spongent [14] results in achieving a significantly more compact design by keeping the round function very simple and hence reducing the logic size close to the smallest theoretically.

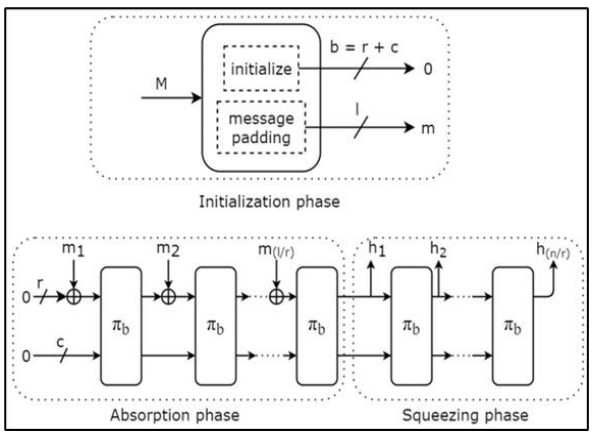

Fig. 1. Sponge-based construction.

The proposed architecture focuses on reducing the SPONGENT-88 variant which is suitable for resource constraint devices to have better throughput and area occupied on both FPGA and ASIC implementation. We have explored the fair balance in the sequential and combinational logic with horizontal data path folding to realize the architecture. As on FPGA, the number of slices occupied on it depends on the distribution of both sequential and combinational logic, combinational logic synthesizes as Look-up-table(LUT) and sequential logic synthesizes as Flipflops(FF). The proposed design uses only one STATE register, and the rest of the logic required to update the register is realized using combinational logic with respective internal modules.

IV. Proposed Architecture for SPONGENT-88

SPONGENT algorithm has five parameters (*n*, *b*, *c*, *r*, *R*) to instantiate a specific Hash function. These parameters vary for different variants of the algorithm. The parameters for SPONGENT-88 are $n = 88$, $b = 88$, $c = 80$, $r = 8$, and $R = 45$ where n is the output, b is the internal state size or width, r is the rate and c is the capacity and R is the number of permutation rounds.

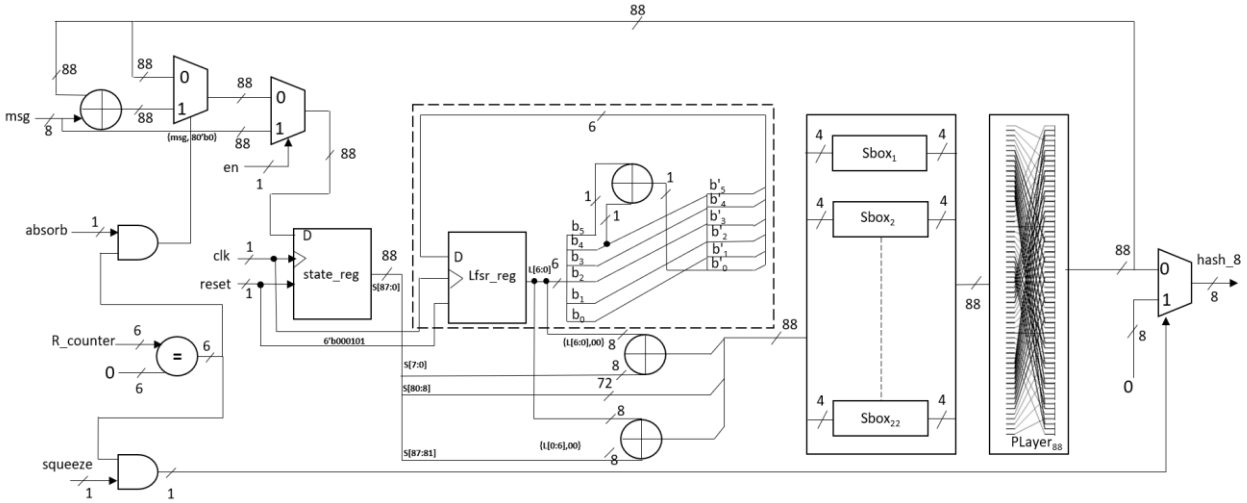

Fig. 2. The proposed architecture for SPONGENT-88/80/8 lightweight sponged-based Hash algorithm

The architecture developed for SPONGENT-88 is a serialized iterative architecture using a single module for the permute block which is used for the transformation of the b bits π_{88}. Initially, all the bits of the STATE are zeros, where the state is a register. In the first round, the input message block of r bits is XOR-ed with the value in the state register in the permute function. In each round, the STATE register is then XOR-ed with the contents of an LFSR, reversed bits of LFSR, and then processed by the substitution and permutation layers as seen in Fig. 2. The proposed architecture makes use of a single-clocked state register and a single-clocked linear feedback shift register, the substitution byte, and the permutation modules are asynchronous, i.e. they change once any change in the input is triggered. Once the reset signal is asserted the state register is initialized to zero and Linear feedback shift register (LFSR) register is initialized to 6-bit binary value 000101. A counter module is implemented which is clocked for each active round of the SPONGENT-88 algorithm, it is an up-down counter. This architecture cascades both the absorption and squeeze phases. After the reset signal goes low, enable signal is asserted to start the permute function, where the 8 bits of the message are concatenated with the zeros of the state register. As the enable signal goes low, the absorption signal is asserted and the contents of the state register are processed by XOR-ing the contents of the Linear feedback shift register with the most significant bits of the state register and reversed contents of the Linear feedback shift register with the least significant bits of the state register. The linear feedback shift register gets updated for every clock cycle. For each clock cycle the b [4] and b [5] bit of the shift register gets XOR-ed and the contents of the register shift by 1 position appending the XOR-ed bit at the MSB of the register. Then the state register is processed through the substitution byte module which consists of 22 Sbox modules called parallelly to create confusion in the bits of the state register followed by 88-bit permutation module to provide diffusion of the state register bits. The Sbox module is implemented using lookup tables, i.e. using mux's, and permutation is simple hard wiring of the state register bits. This whole process comprises a single round of permute function, which repeats for 45 rounds for the same message block of size 8 bits only. Each message block of 8 bits is subsequently XOR-ed with the state register. The absorb signal remains high until all the message bits in terms of r-bit rate are processed, where r = 8-bit. Then the absorb signal goes low and the squeeze signal is asserted and the state register now undergoes the permute function for 45 rounds for producing a Hash value of 8 bits. This process continues until the squeeze phase produces a Hash value of 88 bits. In the squeeze phase, when the counter rolls from the value 44 to 0, the 8-bit Hash value is tapped from the permutation layer. Similarly the same repeats for 88 bits of the Hash value to be generated.

The architecture for the proposed Spongent-88 with FSM is designed with the control signals absorb, squeeze,

and enable with four states in the FSM. The states are CRTL_INIT, CRTL_ABSORB, CRTL_SQUEEZE, PERMUTE, and CRTL_DONE. It was not found necessary to have a state for padding the message bits, as the architecture pads only a byte of information. Hence this is taken care of in the CRTL_INIT state. The FSM is written to produce a Hash digest of 88-bit from a message block of 256-bit to have similar comparisons with the works reported in the literature. In the CRTL_ABSORB state, all the message bits inclusive of padding are absorbed in a block size of *r* bits, and each *r* bits undergo the PERMUTE state for 45 rounds, as it is being determined for the Spongent-88 variant by the original work [5]. As seen in Fig. 3 it gives the block level architecture Spongent-88 core with the controller. In the CRTL_SQUEEZE state, the absorbed message bits further go through the PERMUTE state for 45 rounds at the end of it yields an 8-bit Hash digest, which keeps updating the 88-bit Hash digest register in the controller module until the entire 88-bit Hash digest is formed.

The state diagram for the FSM is as seen the Fig. 4, once the enable and absorb signal goes high the state moves from CRTL_INIT to CRTL_ABSORB the control signal absorb remains high as long the message bits inclusive of padding get absorbed until the value of the meassage_block reaches 33 (8-bits per block × 33 equates to 264-bit, inclusive of the padding bits) as seen in the state diagram the control remains in the PERMUTE state depending on the value of the round counter. The control signal absorb is made '0' and the squeeze signal is asserted, now the control of the finite state machine goes from CRTL_ABSORB to CRTL_SQUEEZE the control remains absorbed until the value of the Hash_length reaches 11 (8-bit Hash output from PREMUTE state × 11 equates to 88-bits Hash output) the control remains in PERMUTE state depending on the value of the round counter until all the 88-bit Hash digest is produced and updated in the CRTL_SQUEEZE state. Once the 88-bit Hash digest is obtained state moves from CRTL_SQUEEZE to CRTL_DONE and then again to the CRTL_INIT state.

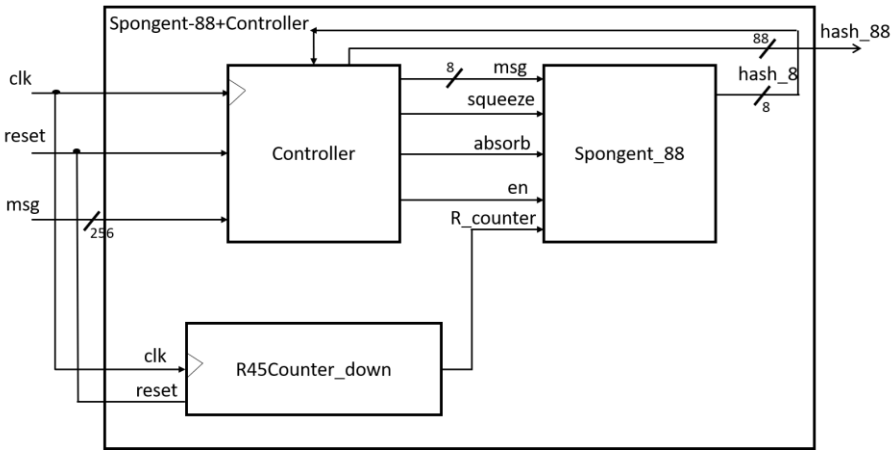

Fig. 3. Proposed architecture for Spongent-88 with Controller.

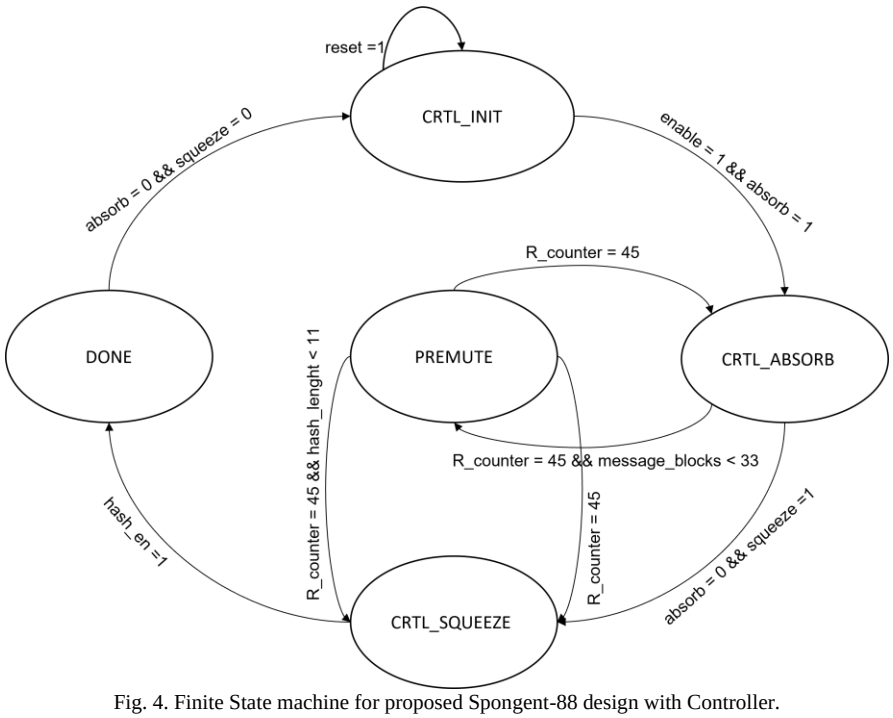

Fig. 4. Finite State machine for proposed Spongent-88 design with Controller.

V. Results and Discussions

The proposed architecture for the SPONGENT-88 sponge-based lightweight algorithm is realized on FPGA and ASIC platforms for a comparison of reported works on both domains.

Section A discusses the implementation results obtained on the FPGA platform for LUT-4 and LUT-6 technology target boards. The proposed architecture was also realized on the ASIC platform with TSMC180nm technology is discussed in Section B.

A. FPGA Implementation

The proposed architecture for Spongent-88 was implemented using a minimal number of internal state registers for better throughput efficiency with a trade-off on memory. The design was constructed using modules of Permute (π_b), Sbox, and Player. An FSM was written with control signals for the transition of the different phases of the algorithm from initialization, absorption, and squeezing. The description of the target platform and metrics used for measuring the efficiency and implementation results are as follows.

The proposed architecture was simulated and synthesized for Xilinx FPGA (LUT-4 and LUT-6 Technology) using ISE Design Suite 14.2 and results were captured after the place and route phase. Spartan 3(xc3s50-5cp132) board and Virtex4 (xc4vfx12-12sf363) was chosen for the implementation under LUT-4 technology. The design was implemented for a frequency constrained to 100kHz. For LUT-6 technology Spartan6 (xc6slx16-3csg324) and Artix A7-100T was chosen. A board-specific constraint file was generated for the targeted FPGA for each of the boards. The implementation of the proposed architecture on the Artix-A7-100T board was realized using Vivado design suite 2018.1. The implementation architecture of the proposed design focuses on metrics: area, throughput, and power. The results for the above metrics are tabulated in Table I to Table IV in terms of area: flip-flops, lookup table, and slice, performance: throughput (unconstrainted), throughput (constrained at 100 kHz), maximum frequency F_{max}, and power: static and dynamic. The other two derived metrics for area and performance are throughput/slice (unconstrainted), throughput/slice[a] (constrained at 100 kHz), and energy consumption: total ENE[a] and ENE[a]/bit. Throughput is calculated as Thru = $B_{\_size} \times F_{max}$/LAT and energy consumed is calculated as ENE = $\text{Total}_{_power} \times$LAT/(constrained frequency), where F_{max} is the maximum frequency reported, *LAT* is the latency cycles required to produce the output digest for an input size of $B_{\_size}$ which is 256-bit and represents the input block size. The power and energy consumption of the proposed architecture for the LUT-4 and LUT-6 technology are tabulated in Table II and Table IV. Power analysis is performed at 100kHz clock (reasonable for applications targeted by lightweight Hash) using the Xpower analyzer tool on ISE 14.2 design suite for the proposed architecture.

TABLE I: Resource Utilization and Performance of SPONGENT-88 Algorithm on LUT- 4 FPGA Boards

Work	Design	Platform	Resource Utilization			LAT (Cycles)	Max. Freq. (MHz)	Thru (Mbps)	Thru/Slice (Mbps/Slice)	Thru[a] (Kbps)	Thru/Slice[a] (Kbps/Slice)
			FF	LUT	Slice						
Lara-Nino *et al.* [23]	SPC-88	SP3	104	143	74	1980	227	29.32	0.39	12.93	0.17
This Work	**SPC - 88**		**94**	**143**	**74**	**1980**	**291**	**35.55**	**0.48**	**12.93**	**0.12**
This Work	**SPCFSM - 88**		**216**	**301**	**210**	**1980+**	**102**	**13.18**	**0.10**	**12.93**	**0.05**
This Work	**SPC - 88**	VT4	**94**	**143**	**74**	**1980**	**700.2**	**90.53**	**1.22**	**12.93**	**0.17**
This Work	**SPCFSM - 88**		**216**	**301**	**210**	**1980+**	**342.7**	**44.30**	**0.21**	**12.93**	**0.05**

TABLE II: Power and Energy Consumption of SPONGENT-88 Algorithm on LUT- 4 Technology Target Boards

Work	Design	Platform	Power[a] (mW)			LAT (Cycles)	ENE[a] (uJ)	ENE[a]/bit (uJ/bit)
			Static	Dynamic	Total			
Lara-Nino *et.al* [23]	SPC- 88	SP3	27.25	0.81	28.06	1980	555.59	2.17
This Work	**SPC- 88**		**27.25**	**0.68**	**27.93**	**1980**	**571.03**	**2.23**
This Work	**SPCFSM - 88**		**27.25**	**1.92**	**29.17**	**1980+**	**577.56**	**2.25**
This Work	**SPC- 88**	VT4	**164.97**	**10.29**	**175.26**	**1980**	**3470.14**	**13.55**
This Work	**SPCFSM - 88**		**164.97**	**17.89**	**182.86**	**1980+**	**3620.62**	**14.14**

SPC- 88: SPONGENT-88 Core only; SPCFSM-88: SPONGENT-88 Core with FSM
SP3 - Spartan 3 xc3s50-5cp132; VT4 - Virtex 4 xc4vfx12-12sf363; [a] Frequency constrained at 100kHz

TABLE III: Resource Utilization and Performance of SPONGENT-88 Algorithm on LUT- 6 FPGA Boards

Work	Design	Platform	Resource Utilization			LAT (Cycles)	Max. Freq. (MHz)	Thru (Mbps)	Thru/Slice (Mbps/Slice)	Thru[a] (Kbps)	Thru/Slice[a] (Kbps/Slice)
			FF	LUT	Slice						
Lara-Nino *et al.* [23]	SPC- 88		104	71	20	1980	302	39.04	1.90	12.93	0.64
Jngk *et al.* [24]	SPC- 88		-	-	26	1980	309	39.95	1.53	12.93	0.49
This Work	**SPC-88**		**94**	**71**	**20**	**1980**	**358**	**46.28**	**2.30**	**12.93**	**0.64**
This Work	**SPCFSM-88**		**211**	**203**	**98**	**1980**	**156.78**	**20.29**	**0.20**	**12.93**	**0.13**
This Work	**SPC- 88**	AT7	**94**	**69**	**20**	**1980**	**404.53**	**52.23**	**2.37**	**12.93**	**0.58**
This Work	**SPCFSM-88**		**211**	**147**	**57**	**1980+**	**208.07**	**26.90**	**0.47**	**12.93**	**0.21**

TABLE IV: POWER AND ENERGY CONSUMPTION OF SPONGENT-88 ALGORITHM ON LUT-6 TECHNOLOGY TARGET BOARDS

Work	Design	Platform	Power[a] (mW)			LAT (Cycles)	ENE[a] (uJ)	ENE[a]/bit (uJ/bit)
			Static	Dynamic	Total			
Lara-Nino *et al.*[23]	SPC- 88	SP6	19.91	1.28	21.19	1980	419.56	1.64
This Work	**SPC- 88**		**19.91**	**0.46**	**20.36**	**1980**	**403.12**	**1.57**
This Work	**SPCFSM - 88**		**20.01**	**1.05**	**22.02**	**1980+**	**440.40**	**1.72**
This Work	**SPC- 88**	AT7	**14.20**	**0.23**	**14.43**	**1980**	**285.71**	**1.11**
This Work	**SPCFSM - 88**		**15.30**	**0.89**	**16.19**	**1980+**	**320.56**	**1.25**

*SPC- 88: SPONGENT- 88 Core only; *SPCFSM - 88: SPONGENT-88 Core with FSM
*SP6 - Spartan 6 xc6slx16-3csg324; *AT7 - Artix A7-100T; [a] Frequency constrained at 100Khz

The static/quiescent, dynamic, and total power utilized by the proposed architecture is calculated using a set of design files with Xpower analyzer tool. Where static power is the power due to leakage current and dynamic power is the power consumed by the logic, load capacitance, and switching frequency of the design under implementation.

The percentage increase/decrease is calculated using the formula:

%increase/decrease= [|difference(intial_reported_value – new_reported_value)| / intial_reported_value] × 100.

The proposed architecture has been realized in two designs: a) Spongent-88 Core and b) Spongent-88 with the FSM, for an input block size of 256 bits which includes the round counter module in the latter. The metrics for both resource utilization and performance are obtained for each of the modules and compared with the reported works in the literature as given in Table I to Table IV.

As seen in Table I, our proposed design on Spartan-3 board uses only 94 flipflops which is 9.6% lesser than Lara-Nino *et al.* [23] which uses 104 flipflops. Although the look-up-tables utilized are the same as in work [23], because of fewer flipflops, the throughput has increased. An improved maximum operating frequency of 291Mhz is been reported. On FPGA the total area occupied is measured in terms of the number of slices occupied. Throughput/slice being the FOM (Figure of Merit), the proposed design reports increased by 18.75% in comparison with the work [23]. On Virtex 4 board the proposed design achieves a maximum operating frequency of 700.2MHz. The design is also realized with FSM for an input block size of 256-bit, where the design reports a maximum operating frequency of 102MHz on Saprtan-3 board and 342.7MHz on Virtex-4 board.

The power consumption in terms of dynamic and quiescent power of the proposed design on Saprtan-3 and Virtex-4 of LUT-4 technology is tabulated in Table II. As discussed above our design uses lesser flipflops which has resulted in better consumption of dynamic power and a decrease in the total power consumed by 4.6% in comparison with work [23], which is related to the switching activity of the flipflops.

Table III gives the tabulation of the area and performance metrics for the proposed design on Spartan - 6 and Aritx-7 board. As the flipflops remain the same on LUT-4 technology board which is 10.6% lesser than Lara-Nino *et al.* [23]. An improved maximum operating frequency of 358Mhz is been reported. As Throughput/slice is the FOM(Figure of merit), the proposed design reports increased by 21% in comparison with the work [23] and 50.3% with the Jungk *et al.* [24]. On Artix-7 board the proposed design achieves a maximum operating frequency of 404.53 MHz. The design is also realized with FSM for an input block size of 256-bit, where the design reports a maximum operating frequency of 158.78 MHz on Saprtan-6 board and 208.07 MHz on Artix-7 board.

The power consumption of the proposed design on Saprtan-6 and Artix-7 of LUT-6 technology is tabulated in Table IV. As our design uses lesser flipflops which has resulted in better consumption of dynamic, which is related to the switching activity of the flipflops. And reports a decrease in total power by 3.91% in comparison with Lara-Nino *et al.* [23].

In the work Jungk *et al.* [24] the architecture uses FIFO for feeding the 8-bit input and a FIFO for reading the 8-bit output, along with a STATE register. Hence increasing the area utilization and number of slices occupied. When compared with the architecture by Lara-Nino *et al.* [23], the output is taken from the STATE register after being updated and keeps the assert signals of absorption and squeeze phase outside the architecture.

Whereas even our proposed algorithm uses a single STATE register which is clocked and updated for the subsequent permute function rounds of the algorithm either for absorption or squeeze phase. The output is taken from the permutation layer at the end of 45 rounds of the squeeze phase respectively. The proposed architecture cascades absorption and squeeze phases of the algorithm in the same architecture by relevant control signals and hence making the throughput/slice of the proposed architecture better than the works in [23, 24] by 21% and 50.03%, respectively on LUT-6 technology board.

B. ASIC Implementation

The proposed architecture was realized on the ASIC domain to determine the implementation costs on silicon which proposes a different challenge when compared to realizing it on FPGA. The analysis is carried out by implementing the proposed architecture on different technology nodes and standard cell libraries and studying the behavior of implementation cost for the sequential and combinational logic of the proposed architecture for the SPONGENT-88 lightweight Hash algorithm.

The area occupied by the design on silicon is determined using the metric gate-equivalence which accounts for the physical area required to implement it on the circuit/board and is measured in μm^2, one gate-equivalence is defined as the area occupied by a 2-input NAND gate on the respective technology node or standard cell library. The gate-equivalence is a technology-independent metric to measure the size occupied by the design.

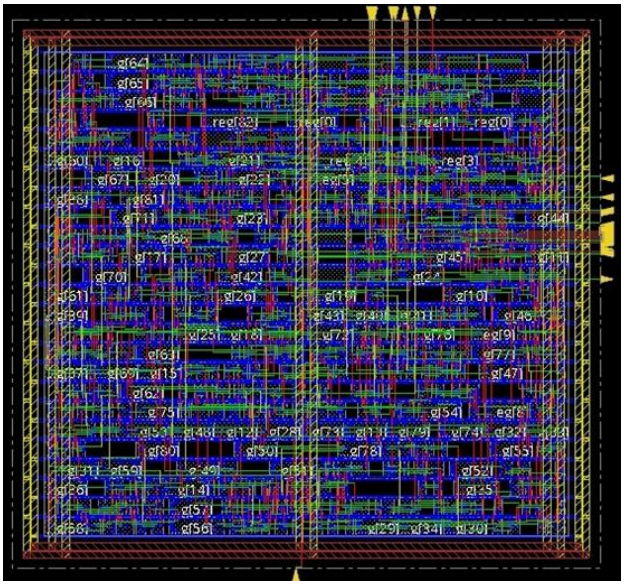

Fig. 5. Layout of proposed architecture on TSMC180nm node.

Hardware implementations of lightweight cryptographic primitives are classified as ultra-lightweight, low-cost, and lightweight occupying up to 1000, 2000, and 3000 gate equivalence respectively. The authors in [2] classify the lightweight cryptographic primitives in two categories as ultra-lightweight cryptography and ubiquitous cryptographic, where the latter deals with primitives fulfilling a unique purpose while satisfying specific and narrow constraints and the former encompass more versatile algorithms/primitives both in terms of implementation trade-offs and functionality.

A lightweight Hash algorithm's footprint depends upon two factors, first the number of state bits (inclusive of the key schedule for block cipher based) and the size occupied by the functional and control logic used to realize a round function. The proposed architecture has a serialized approach with a data path size of 88 bits, it is designed to utilize lesser flip-flops and has a good tradeoff between the sequential and combinational logic required to realize the Spongent-88 sponge-based lightweight Hash algorithm. The proposed design has reported 1204 GE with 45 cycles required for each block of data on UMC180nm technology in comparison to 1232 GE which was originally reported work of Spongent-88 with the same data path [14] on a similar technology node. The layout of the proposed architecture is seen in Fig. 5.

The results obtained on TSMC180nm technology are seen in Table V which gives the detailed area occupied for the individual modules Permute, Sbox, and Player. Table VI gives the gate-equivalence obtained from TSMC180nm for SPONGENT-128 and SPONGENT-224 of the proposed architecture. Therefore, this work has reported the 1204 GE occupied for the 88-bit data path on TSMC180nm technology, whereas the original work [16] reports 1232 GE on a similar technology node, i.e. UMC180, the proposed work has a 2.27% decrease in the area occupied.

The proposed architecture is also realized for SPONGENT-128 and SPONGENT-224 keeping the architecture similar with an increase in the number of instantiations of Sbox, Permutation bits, and internal State register for each of the variants respectively. The proposed architecture for SPONGENT-128 and SPONGENT-224 have security bounds that are comparable with SHA-1 and SHA-2 [8] NIST recommended Hash algorithms. The gate-equivalence of the latter variants of SPONGENT is tabulated in Table VI, where the footprint is reduced by 3.5% and 6.4% for SPONGENT-128 and SPONGENT-224 respectively.

To have a familiar comparison of the proposed SPONGENT-88 architecture under 1000 GE with the original work, the proposed work has also implemented the algorithm in a serialized architecture, where a single S-box module is utilized instead of 22 instances of the S-box, keeping the rest of the architecture the same. This work has achieved 819 GE on TSMC180 technology node.

TABLE V: AREA OF THE PROPOSED WORK ON TSMC180NMNODE

Design	Data path (bits)	Module		Gate-equivalence (GE)
SPONGENT- 88/80/8	88	Permute		16.66
		S-box	1- instance	27.00
			22- instance	594.00
		Permutation layer (Player)		0.00
		Glue Logic		593.34
		Total		(Sequential -537.67) + (Combinational - 599.01 + Inverter - 68.00) = 1204

TABLE VI: AREA OF THE PROPOSED WORK ON TSMC180NM TECHNOLOGY FOR SPONGENT-128 AND SPONGENT-224 VARIANT

Design	Data path (bits)	Module		Gate-equivalence (GE)	Total (GE)
SPONGENT-128/128/8	136	Permute		45.12	1790
		S-box	34- instance	918.00	
		Glue logic		826.88	
SPONGENT-128/128/8[14]		-		-	1855
SPONGENT-224/224/16	240	Permute		121.31	2995
		S-box	60-instance	1620	
		Glue logic		1253.69	
SPONGENT- 224/224/16[14]		-		-	3203

The power consumption for a frequency constrained at 100kHz for the proposed SPONGENT-88, 128, 224 variant as computed on the ASIC platform on the genus tool is tabulated in Table VII. As the power computed is on different technology nodes of the original work [14]. Since power consumption cannot be compared across different technology nodes. The proposed work consumes lesser power on higher technology nodes as referred to in Table VIII. Due to the availability of all the technology node-dependent files required to generate the cells, power, and timing reports, the proposed work was carried out on the TSMC180 node. The work [25] also implements the Spongent-88 algorithm in its full data path of 88 bits. The proposed work reports a better reduction in the area by 15% and in power by 46% in comparison with the latter one as observed in Table VII. The lightweight hash algorithms discussed in the Related works section have been classified on gate-equivalence of hardware implementation as lightweight, low-cost, and ultra-lightweight cryptographic primitives in Table VIII, Table IX, and Table X with gate equivalence between less than 1000 GE, 2000-3000 GE, and 1000-2000 GE respectively, along with the security bounds on pre-image, second pre-image, and collision attacks for the each of the algorithms. The proposed algorithms for Spongent variants, SPONGENT-88 with a data path of 4-bit, SPONGENT-128 with a data path of 136-bit, and SPONGENT-224 with a data path of 240-bit. Classify under the lightweight, low-cost, and ultra-lightweight cryptographic primitives.

Under lightweight primitive, our proposed design for SPONGENT-88 with 4-bit data path achieves increased throughput by 46% in comparison with work [25].

TABLE VII: POWER CONSUMPTION OF THE SPONGENT FAMILY FOR 88/128/224 VARIANT WITH THE PROPOSED SPONGENT ARCHITECTURE FOR A FREQUENCY OF 100KHZ

Hash Algorithm	Security			Hash (bit)	Datapath	Cycles	Process (µm)	Power (µW)
	Pre.	2nd Pre.	Col.					
Spongent-88/80/8 [14]	80	40	40	88	88	45	0.13	1.57
Spongent-88/80/8 [25]	80	40	40	88	88	45	0.18	1.5
This Work Spongent-88	**80**	**40**	**40**	**88**	**88**	**45**	**0.18**	**1.37**
Spongent-128/128/8 [14]	120	64	64	128	136	70	0.13	3.58
Spongent-128/128/8 [25]	120	64	64	128	136	70	0.18	12.85
This Work Spongent-128	**120**	**64**	**64**	**128**	**136**	**70**	**0.18**	**3.01**
Spongent-224/224/16 [14]	208	112	112	224	240	120	0.13	5.97
Spongent-224/224/16 [25]	208	112	112	224	240	120	0.18	22.98
This Work Spongent-224	**208**	**112**	**112**	**224**	**240**	**120**	**0.18**	**4.91**

TABLE VIII: COMPARISON OF DIFFERENT HASH ALGORITHMS FOR GE< 1000

Algorithm	Parameters				Security Bounds			Data path	Cycles	Process (µm)	Area (GE)	Thru (kbps)
	n	*b*	*C*	*r*	Pre	2nd Pre	Col					
PHOTON [11]	80	100	80	20	64	40	40	4	708	0.18	865	2.82
Spongent [14]	80	88	80	8	80	40	40	4	990	0.13	738	0.81
LHash [13]	80	96	80	16	64	40	40	4	666	0.13	817	1.44
Spongent [14]	80	88	80	8	80	40	40	4	990	0.18	759	*
Spongent-88 [25]	80	88	80	8	80	40	40	4	990	0.18	967	1.5
This Work: Spongent-88	**88**	**88**	**80**	**8**	**80**	**40**	**40**	**4**	**990**	**0.18**	**819**	**0.81**

n: Hash digest, *b*: internal state size, *C*: capacity, *r*: rate, Pre: preimage, 2nd – Pre: Secondary preimage, Col: collision, * not reported

TABLE IX: COMPARISON OF DIFFERENT HASH ALGORITHMS FOR 1000>GE< 2000

Algorithm	Parameters				Security Bounds			Data path	Cycles	Process (µm)	Area (GE)	Thru (kbps)
	n	*b*	*C*	*r*	Pre	2nd Pre	Col					
PHOTON [11]	128	144	128	16	112	64	64	4	996	0.18	1122	1.16
	80	80	20	16	64	40	40	20	132	0.18	1168	12.15
	160	160	36	36	124	80	80	4	1132	0.18	1396	2.70
	128	128	16	16	112	64	64	24	156	0.18	1708	10.26
	224	224	32	32	192	112	112	4	1716	0.18	1735	1.86
Spongent [14]	88	88	80	8	80	40	40	88	45	0.13	1127	17.78
	128	128	128	8	120	64	64	4	2380	0.13	1060	0.34
	160	160	160	16	144	80	80	4	3960	0.13	1329	0.40
	224	224	224	16	208	112	112	4	7200	0.13	1728	0.22
	88	88	176	88	88	44	88	4	8910	0.13	1912	0.99
	256	256	512	256	256	128	256	4	9520	0.13	1950	0.17
U-Quark [12]	128	136	128	8	120	64	64	1	544	0.18	1379	1.47
D-Quark [12]	160	160	128	8	144	80	80	1	704	0.18	1702	2.27
LHash [13]	128	128	120	8	120	60	60	4	882	0.18	1028	0.40
	128	128	120	8	120	60	60	4	72	0.18	1200	4.94
	128	128	112	16	96	56	56	4	882	0.18	1028	1.21
	128	128	112	16	96	56	56	4	72	0.18	1200	14.81
Keccak-f [8]	80	100	80	20	60	40	40	4	800	0.18	1250	1.50
DM-Present [19]	80	-	64	-	64	32	64	4	547	0.18	1600	14.63
	128	-	128	-	64	32	64	4	559	0.18	1886	22.90
Spongent-128 [14]	128	128	128	8	120	64	64	4	70	0.18	1855	*
This Work: Spongent-128	**128**	**128**	**128**	**8**	**120**	**64**	**64**	**136**	**70**	**0.18**	**1790**	**11.42**

n: Hash digest, *b*: internal state size, *C*: capacity, *r*: rate, Pre: preimage, 2nd – Pre: Secondary preimage, Col: collision, * not reported

TABLE X: COMPARISON OF DIFFERENT HASH ALGORITHMS FOR 2000 >GE< 3000

Algorithm	Parameters				Security Bounds			Data path	Cycles	Process (µm)	Area (GE)	Thru (kbps)
	n	*b*	*C*	*r*	Pre	2nd Pre	Col					
Spongent [14]	128	128	256	128	128	64	128	4	18720	0.13	2641	0.68
	160	160	160	16	144	80	80	176	90	0.13	2190	17.78
	224	224	224	16	208	112	112	240	120	0.13	2903	13.33
	224	224	224	112	208	112	112	4	14280	0.13	2371	0.78
Photon [11]	160	160	36	36	124	80	80	28	180	0.18	2117	20.00
	224	224	32	32	192	112	112	32	204	0.18	2786	15.69
	256	256	32	32	224	128	128	8	996	0.18	2177	3.21
u-Quark [12]	128	136	128	8	120	64	64	8	68	0.18	2392	11.76
d-Quark [12]	160	160	128	8	114	80	80	8	88	0.18	2819	18.18
s-Quark [12]	224	224	128	8	192	112	112	1	1024	0.18	2296	3.13
DM- PRESENT [19]	80	-	64	-	64	32	64	64	33	0.18	2213	242.42
	128	-	64	-	64	32	64	128	33	0.18	2530	387.88
H- PRESENT [19]	128	-	64	-	128	64	64	8	559	0.18	2330	11.45
Keccak-f [8]	128	200	128	72	64	64	64	8	900	0.13	2520	8.00
GLUON [10]	64	-	-	-	128	-	64	8	66	*	2071	12.12
	80	-	-	-	160	-	80	16	50	*	2799	32.00
Spongent- 224 [14]	224	224	224	16	208	112	112	240	120	0.18	3203	*
This Work: Spongent-224	**224**	**224**	**224**	**16**	**208**	**112**	**112**	**240**	**120**	**0.18**	**2295**	**13.33**

n: Hash digest, *b*: internal state size, *C*: capacity, *r*: rate, Pre: preimage, 2nd – Pre: Secondary preimage, Col: collision, * not reported

VI. CONCLUSION

This work has evaluated the hardware implementation of the proposed architectures for the SPONGENT-88, SPONGENT-128, and SPONGENT-224 variant of the SPONGENT lightweight Hash algorithm for the proposed architecture. The SPONGENT-88 is been compared with state-of-art works done on the FPGA platform. In the ASIC domain, the proposed work is been carried out on TSMC180nm technology and has been compared to the original work reported [14]. To increase the throughput of the architecture on FPGA a fair balance of Look-up-tables and Flipflops is achieved in the proposed architecture. Since throughput/slice is the FOM (Figure of Merit) for measuring the performance of the proposed design and the work has achieved 50.03% and 21% more throughput/slice in comparison with the works Lara-Nino *et al.* [23] and Jungk *et al.* [24] on Spartan-6 technology board. Original work [14] reports 1232 GE and the proposed architecture on the similar technology occupies 1204 GE with a reduction of 2.27% in the area. Recent works also [26] show interest in SPONGENT algorithm as an authentication primitive in blockchain technology. This work can be further used to design and develop a lightweight authenticated encryption or authenticated permutation encryption engine/module at end nodes with both data privacy/confidentiality with authentication at end nodes [27].

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

Nalini C. Iyer was instrumental in the study of lightweight authentication primitives, identifying the algorithm and articulating the findings and analysis. Heera G. Wali carried out the optimized implementation and analysis of the proposed work.

REFERENCES

[1] S. Kerckhof, F. Durvaux, François and C. Hocquet *et al.* "Towards green cryptography: A comparison of lightweight ciphers from the energy viewpoint," *Lecture Notes in Computer Science*, vol. 7428, pp. 390-407, 2017.

[2] M. Rana, Q. Mamun, and R. Islam, "Lightweight cryptography in IoT networks: A survey," *Future Generation Computer Systems*, vol. 129, pp. 77-89, 2022.

[3] K. Tsantikidou and N. Sklavos, "Hardware limitations of lightweight cryptographic designs for IoT in healthcare," *Cryptography*, vol. 6, no. 3, #45, 2022.

[4] S. Abed, R. Jaffal, B. J. Mohd, *et al*., "An analysis and evaluation of lightweight hash functions for blockchain-based IoT devices," *Cluster Comput*, vol. 24, pp. 3065-3084, Jun. 2021.

[5] I. H. Abdulqadder, S. Zhou, D. Zou, I. T. Aziz, and S. M. A. Akber, "Bloc-Sec: Blockchain-based lightweight security architecture for 5G/B5G enabled SDN/NFV cloud of IoT," in *Proc. of 2020 IEEE 20th International Conference on Communication Technology*, 2022, pp. 499-507.

[6] N. A. Gunathilake, A. Al-Dubai, and W. J. Buchana, "Recent advances and trends in lightweight cryptography for IoT security," in *Proc. of 2020 16th Int. Conf. on Network and Service Management*, 2020, doi: 10.23919/CNSM50824.2020.9269083

[7] S. Aruna, G. Usha, P. Madhavan, and M. V. R. Kumar, "Lightweight cryptography algorithms for IoT resource-starving devices," in *Role of Edge Analytics in Sustainable Smart City Development*, G. R. Kanagachidambaresan Ed. 2020, doi: https://doi.org/10.1002/9781119681328.ch8.

[8] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, "Keccak," *Lecture Notes in Computer Science*, vol. 7881, 2013, doi: https://doi.org/10.1007/978-3-642-38348-9_19

[9] S. Badel, N. Dagekin, J. Nakahara Jr., *et al*., "ARMADILLO: A multi-purpose cryptographic primitive dedicated to hardware," *Lecture Notes in Computer Science*, vol. 6225, 2010, doi: https://doi.org/10.1007/978-3-642-15031-9_27

[10] T. P. Berger, J. D'Hayer, K. Marquet, *et al.*, "The GLUON family: a lightweight Hash function family based on FCSRs," *Lecture Notes in Computer Science*, vol. 7374, pp. 306–323, 2012, doi: https://doi.org/10.1007/978-3-642-31410-0_1

[11] J. Guo, T Peyrin, and A. Poschmann, "The PHOTON family of lightweight hash functions," *Lecture Notes in Computer Science*, vol. 6841, pp. 222–239, 2011.

[12] J. P. Aumasson, L. Henzen, W. Meier, M. Naya-Plasencia, "Quark: A lightweight hash," *Lecture Notes in Computer Science*, vol. 6225, pp. 1–15, 2010.

[13] W. Wu, S. Wu, L. Zhang, *et al*., "LHash: A lightweight Hash function," *Lecture Notes in Computer Science*, vol. 8567, pp. 291-308, 2014.

[14] A. Bogdanov, M. Knezevic, G. Leander, *et al*., "SPONGENT: A lightweight hash function," *Lecture Notes in Computer Science*, vol. 6917, pp. 312–325, 2011.
[15] D. N. Gupta and R. Kumar, "Sponge based lightweight cryptographic Hash functions for IoT applications," in *Proc. of International Conference on Intelligent Technologies*, 2021, doi: 10.1109/CONIT51480.2021.9498572
[16] B. Tim, Y. L. Chen, C. Dobraunig, B. Mennink. (2021). Elephant v2 Specification. Submission to NIST LWC Project. [Online]. Available: https://www.esat.kuleuven.be/cosic/elephant/
[17] L. Pyrgas and P. Kitsos, "An 8-bit compact architecture of lesamnta-LW Hash function for constrained devices," in *Proc. of 26th IEEE International Conference on Electronics, Circuits and Systems*, 2019, pp. 743-746.
[18] NIST. (2019). Lightweight Cryptography: Project Overview. [Online]. Available: https://csrc.nist.gov/projects/lightweight-cryptography
[19] A. Bogdanov, G. Leander, C. Paar, *et al.*, "Hash functions and RFID tags: Mind the gap," *Lecture Notes in Computer Science*, vol. 5154, pp. 283-299, 2008.
[20] W. Zhang, Z. Bao, V. Rijmen, and M. Liu, "A new classification of 4-bit optimal S-boxes and its application to PRESENT, RECTANGLE and SPONGENT," *Lecture Notes in Computer Science*, vol. 9054, pp. 494-515, 2015.
[21] B. Ovilla-Martínez, C. Mancillas-López, A. F. Martínez-Herrera, and J. A. Bernal-Gutiérrez, "FPGA implementation of some second round NIST lightweight cryptography candidates," *Electronics* vol. 9, no. 11, #1940, 2020.
[22] M. Tehranipoor, N. Pundir, N. Vashistha, F. Farahmandi, "Lightweight cryptography," in *Hardware Security Primitives.* Springer, Cham, 2023, doi: https://doi.org/10.1007/978-3-031-19185-5_12.
[23] C. A. Lara-Nino, M. Morales-Sandoval, and A. Diaz-Perez, "Small lightweight Hash functions in FPGA," in *Proc. of IEEE 9th Latin American Symposium on Circuits Systems*, 2018. doi: 10.1109/LASCAS.2018.8399948.
[24] B. Jungk, L. R. Lima, and M. Hiller, "A systematic study of lightweight Hash functions on FPGAs," in *Proc. of 2014 International Conference on ReConFigurable Computing and FPGAsc*, 2014, doi: 10.1109/ReConFig.2014.7032493.
[25] B. Rashidi, "Efficient full data-path width and serialized hardware structures of SPONGENT lightweight hash function," *Microelectronics Journal*, vol. 115, 2021, doi: https://doi.org/10.1016/j.mejo.2021.105167
[26] M. Apriani and R. F. Sari, "Performance comparison of spongent and photon hashing algorithms in ethereum-based blockchain system," in *Proc. 7th International Conference on Electrical, Electronics and Information Engineering*, 2021, pp. 564-569.
[27] P. Zhang, "Permutation-based lightweight authenticated cipher with beyond conventional security," *Security and Communication Networks*, vol. 2021, 2021, doi: http://doi.org/10.1155/2021/1468007.

Heera G. Wali is a research scholar and Assistant professor at KLE Technological University, Hubli, Karnataka, India (2016). She has obtained Bachelor's Degree in Electronics and Communication Engineering from B.V. Bhoomaraddi College of Engineering and Technology, VTU Belgaum (2007), and a Master's Degree in VLSI design and testing (2013). Her researches are in the field of image encryption, IoT security, cryptography, lightweight cryptography and its use cases.

Nalini C. Iyer is currently heading School of Electronics and Communication Engineering at KLE Technological University, Hubli, Karnataka, India. She obtained Ph.D. degree in Electronics and Communication (Information Security Algorithm Optimization and Architectures, VTU Belgaum) in 2014.
Her researches are in the fields of. Cryptography, Hardware Security, Embedded systems for autonomous functions (Autonomous Vehicle), Vehicular Communication, and VLSI Design. She is affiliated with IEEE and has served as an invited reviewer. Besides, she is also involved in student associations, curriculum design, and outcome-based education pedagogy development activities at the university.