Secure Hash Algorithm-3(SHA-3) implementation on Xilinx FPGAs, Suitable for IoT Applications

Muzaffar Rao, Thomas Newe and Ian Grout
University of Limerick, Ireland
muhammad.rao @ ul.ie, thomas.newe @ ul.ie, Ian.grout @ ul.ie

Abstract—**Data integrity is a term used when referring to the accuracy and reliability of data. It ensures that data is not altered during operations, such as transfer, storage, or retrieval. Any changes to the data for example malicious intention, unpredicted hardware failure or human error would results in failure of data integrity. Cryptographic hash functions are generally used for the verification of data integrity. For many Internet of Things (IoT) applications, hardware implementations of cryptographic hash functions are needed to provide near real time data integrity checking. The IoT is a world where billions of objects can sense, share information and communicate over interconnected public or private Internet Protocol (IP) networks. This paper provides an implementation of a newly selected cryptographic hash algorithm called Secure Hash Algorithm – 3 (SHA-3) on Xilinx FPGAs (Spartan, Virtex, Kintex and Artix) and also provides the power analysis of the implemented design. An FPGA is the best leading platform of the modern era in terms of flexibility, reliability and re-configurability. In this implementation the core functionality of SHA-3 is implemented using LUT-6 primitives and then these primitives are instantiated for the complete implementation of SHA-3. The Xilinx Xpower tool is used for power analysis of the implemented design. This implementation can be used with IoT applications to provide near real time data integrity checks.**

Index Terms— FPGA, IoT, SHA-3, Data Integrity

1. INTRODUCTION

In a public network, data flow between the IoT applications can be visible to a number of nodes on the network. Although data can be secured using encryption there may be a chance of data alteration on the network, whether the data is encrypted or not. This alteration can be catastrophic for the applications running at the destination node and can often lead to incorrect responses. For example, in a fund transfer if the hacker alters random pieces of data this may lead to incorrect funds being transferred, or even transfer to an incorrect account. Data can be altered due to many reasons; malicious intention, unpredicted hardware failure and human error in the network. With data integrity implemented data alteration can be detected, if the data integrity check is applied at the receiver side. This check can detect incorrect data transfer and therefore incorrect transactions or operations can be prevented.

The authors would like to thank the Erasmus Mundus STRoNGTiES (Strengthening Training and Research through Networking and Globalization of Teaching in Engineering Studies) program for providing funding that has facilitated the completion of this work.

Hash functions can be used for the verification of data integrity. This is a one-way deterministic procedure whose input is an arbitrary block of data and whose output is a fixed-size bit string, which is known as the hash value. The data to be encoded is called the message, and the hash value is called the message digest. In short, a message digest is a fingerprint of the data. If the data changes the fingerprint changes. In addition, if a secret is used in the process to generate the hash (called a HMAC) then no one can predict the corresponding digest of the data without knowing the secret and also the content of the hashed data cannot be determined from the hash value. This is why it is called a one-way procedure.

The hash of the data is calculated and appended to the data. When the message arrives at its destination (in the case of data transfer) or is retrieved (in the case of data storage), the hash is recalculated from the data and compared to the hash that was appended to the original message. If the values do not match, then it means that the data has been altered. Fig. 1 shows how a HMAC is generated and compared using a shared secret K. A HMAC, a Hash generated Message Authentication Code can be used to verify both integrity of a message and its source.

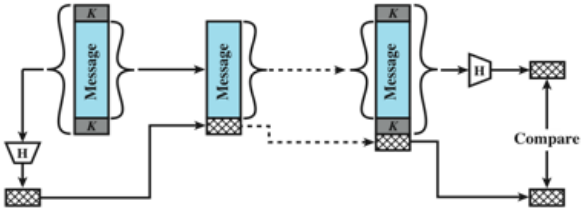

Figure 1. Secure Hash generation and comparison (HMAC).

Commonly used Hash functions are SHA-1, SHA-256, SHA-512, RIPEMD, MD4 and MD5. In previous years Cryptanalysis of these algorithms has found serious vulnerabilities [1][2][3]. Although no attacks have yet been reported on the SHA-2 variants, but due to their algorithmically similarities to SHA-1, there are fears that SHA-2 could also be cracked in the near future. The National Institute of Standards and Technology (NIST), USA announced the SHA-3 Contest in Nov 2007 [4]. This contest was to result in a new and secure cryptographic hash algorithm. This competition ended on 2nd October'12, after the announcement of Keccak, one of the finalists of SHA-3 competition, as the winner for the title of SHA-3[5].

For the IoT applications, hardware implementations of cryptographic hash algorithms are needed to provide high speed and near-real time results. ASICs and FPGAs are the

two hardware platforms that can be used for these implementations. FPGAs offer numerous advantages for algorithm implementations over ASICs, such as: reliability, flexibility, low cost, rapid time-to-market and long-term maintenance [6].

Power efficiency is a critical issue in IoT applications, since the devices connected in IoT are often not connected to power directly and have to operate using energy harvesting sources or a single battery for several years without maintenance or battery replacement. So, the Xilinx Xpower tool is used for power analysis of the implemented design to have an idea about the power consumption on different Xilinx FPGAs.

In this paper the LUT-6 (Look-Up-Table -6) primitives of FPGA are used for the SHA-3 implementation. The LUT-6 primitive can be used to implement any function of 6-inputs. The logical function of the SHA-3 core is divided into two parts **(1)** 5- input *XOR* logic **(2)** 5-input *Chi* logic. The *chi* logic consist of *XOR*, *NOT* & *AND* logical operation. Both of these 5-input logics are stored in a single LUT-6 primitive where the remaining input of the LUT-6 primitive is used as a control signal for the selection between the *XOR* and *chi* logic. For the complete implementation of SHA-3 a number of LUT-6 primitives are used.

Our proposed FPGA architecture is implemented on different Xilinx FPGA platforms. The implementation results on Artix-7 FPGA are better (because of the less static power consumption) for low power operations as compare to other FPGAs. The proposed architecture can be used with any IoT application to provide up-to-date data integrity check using the newly selected hash algorithm i.e. SHA-3. In this paper we also discuss some specific IoT applications to emphasis the importance of data integrity.

The remainder of the paper is organized as follows: Section II gives details about IoT applications, Section III provides FPGA details and in section IV the SHA-3 algorithm is discussed. Section V is about the I/O interface for implementation. The detail of SHA-3 implementation is given in section VI. Power analysis is discussed in section VII and section VIII provides the Implementation results. Finally, the conclusion is presented in section IX.

II. INTERNET OF THINGS APPLICATIONS

The Internet of Things (IoT) is where millions of independent objects can sense, share information and communicate over interconnected public or private networks. These interconnected objects can collect data, analyze it and make decisions on the basis of the collected data. This is the world of the Internet of Things (IoT). In this mixed wired and wireless world there is a significant chance of data alteration at network nodes. So, without data integrity checks one cannot be sure about the originality/source of received data. This dilemma can be largely solved using HMACs. Let's have a look on some of the IoT applications.

One of the important IoT application areas is a Patient Monitoring System. Continuous patient monitoring application requires the use of medical body sensors to monitor vital body conditions such as heartbeat, temperature and sugar levels. This application examines the current state of the patient's health for any abnormalities and can predict if the patient is going to encounter any health problems. The sensor data must be correctly received at the physician's network node as any alteration in original data may provide wrong information about patient's health there by leading to potentially fatal mistreatment of the patient.

Crowd control applications will allow relevant government authorities to estimate the number of people gathering at an event and determine what necessary actions need to be taken during an emergency. The application could be installed on mobile devices and users would need to agree to share their location data for the application to be effective. Using location-based technologies such as cellular, WiFi and GPS, the application could generate virtual "heat maps" of crowds. These maps could be combined with sensor information obtained from street cameras, motion sensors and officers on patrol to evaluate the condition of the crowded areas. Emergency vehicles can also be informed of the best possible routes to take, using information from real-time traffic sensor data. Again we can see that if the application doesn't have a data integrity check, any corrupt data can lead to false information being received with potentially fatal consequences.

The intelligent streetlamp is a network of streetlamps that are tied together in a WAN that can be controlled and monitored from a central point. It captures data such as ambient temperature, visibility, rain, GPS location and traffic density which can be fed into applications to manage road maintenance operations, traffic management and vicinity mapping. With the availability of such real-time data, government can respond quicker to changing environments to address citizen needs. Implementation of data integrity checks is again important for this application.

In short, data privacy and security is very important and is one of the key challenges that need to be addressed before mass adoption of the IoT applications. In the IoT, a lot of data flows autonomously and without human knowledge, so it is necessary that the data received at the receiving node must be correct and unaltered from the transmitted data. To achieve data integrity and source authentication HMACs can be used with a shared secret (Fig. 2).

Figure 2. Data Origin and Integrity check

III. FIELD PROGRAMMABLE GATE ARRAY

FPGAs are made up of an interconnection of logic blocks in the form of a two-dimensional array. The logic blocks consist of look-up tables (LUTs) constructed over simple memories that store Boolean functions. Each LUT has a fixed number of inputs and is coupled to a multiplexor and a Flip-Flop in order to build sequential circuits. Likewise, several LUTs can be combined in order to implement complex functions. The FPGA architecture makes it possible to implement any combinational and sequential circuit, which can range from a simple logic function to a high-end processor [7]. In order to reduce the complexity of designing FPGA systems, Hardware Description Languages (HDL) such as VHDL and Verilog HDL are used. Likewise, the vendor's synthesizer seeks for an optimized arrangement of the FPGA's resources based on the hardware description (particularly the content of the LUTs and interconnectors) during the process of mapping and routing in order to generate a bit stream, which afterwards is loaded into the targeted platform.

Xilinx classifies their FPGAs in three series: Virtex (high-end), Artix (low-power), and Kintex (low-cost). The Spartan series of Xilinx FPGAs is also low cost, but now this series is replaced by Kintex.

In terms of power consumption, FPGA power is divided into two parts: static power (or quiescent power) and dynamic power [8]. Static power is the intrinsic power of the device and cannot be changed. It exists once the chip is powered on, even if there is no activity in the device. It includes transistor leakage, power consumed internally, and power dissipated in external termination resistors. Dynamic power is caused by the switching activity of CMOS transistors. Dynamic power is only consumed when the state of transistors changes, which depends on the specific implementation of the design. A design can consume less power if it is implemented in an appropriate way.

Table I shows that the Artix-7 FPGA has similar static power consumption to the Spartan-3, but it provides a larger space and better optimized blocks. Table I presents the static power consumption and space/LUTs features of the different series of Xilinx FPGAs.

TABLE I. POWER CONSUMPTION OF XILINX FPGAS

Platform	Model	Number of LUTs	Static Power Consumption (mW)
Spartan-3	XC3S200	4,320	41
Spartan-3E	XC3S250E	5,508	51
Spartan-6	XC6SLX100	101,261	67
Virtex-4	XC4VLX200	200,448	1,278
Virtex-5	XC5VLX220	138,240	1,985
Virtex-6	XC6VLX240T	241,152	1,977
Virtex-7	XC7VX330T	326,400	141
Kintex-7	XC7K160T	162,240	74
Artix-7	XC7A100T	101,440	41

IV. SECURE HASH ALGORITHM-3 (SHA-3)

SHA-3[9] is a family of sponge functions characterized by two parameters, the bitrate r and capacity c. The sum, r + c determine the width of the SHA-3 function permutation used in the sponge construction and is restricted to a maximum value of 1600. Selection of r and c depends on the desired hash output value. Ex.: for a 256-bit hash output r = 1088 and c = 512 and for 512-bit hash output r = 576 and c = 1024 is selected. The 1600-bit state of SHA-3 consists of a 5x5 matrix of 64-bit words as shown in Fig. 3.

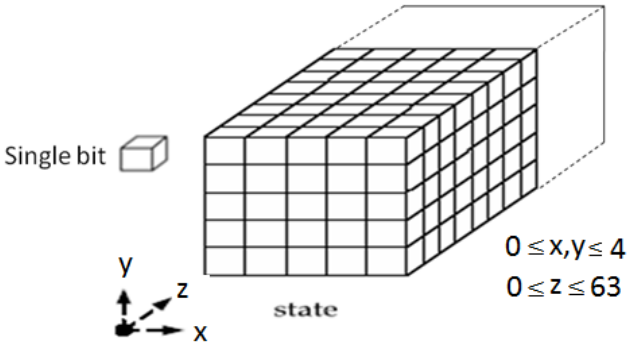

Figure 3. State Matrix (A) of SHA-3

There are 24 rounds in the compression function (Core) of SHA-3 and each round consists of 5 steps, *Theta* (θ), *Rho*(ρ), *Pi*(π), *Chi* (χ) and *Iota* (i) as shown in eq. (1) to (6).

***Theta* (θ) Step:** $(0 \le x,y \le 4)$

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4]; \quad \textbf{(1)}$$

$$D[x] = C[x-1] \oplus ROT(C[x+1],1) ; \quad \textbf{(2)}$$

$$A[x, y] = A[x, y] \oplus D[x]; \quad \textbf{(3)}$$

***Rho* (ρ) and *Pi* (π) Step:** $(0 \le x,y \le 4)$

$$B[y, 2x+3y] = ROT(A[x,y], r[x, y]); \quad \textbf{(4)}$$

Where r [x, y] is the Cyclic Shift Offset

***Chi* (χ) Step:** $(0 \le x, y \le 4)$

$$A[x, y] = B[x,y] \oplus ((NOT\,B[x+1,y]) AND\, B[x+2,y]); \quad \textbf{(5)}$$

***Iota* (i):**

$$A[0, 0] = A[0, 0] \oplus RC ; \quad \textbf{(6)}$$

Where RC is the Round Constant

In the above equations all operations within indices are done modulo 5. In eq. (1) A denotes the state matrix of 1600-bits and A[x, y] denotes a particular 64-bit word in that state. B[x, y], C[x] and D[x] are intermediate variables. Other operations include bitwise *XOR*, *NOT* and the bitwise *AND* logical operation. Finally, ROT denotes the bit wise cyclic shift operation. The constants r[x, y] and RC are cyclic shift offset and round constant respectively and are given in the specifications [9].

The SHA-3 hash function operation consists of three phases: initialization, absorbing and squeezing. Initialization is simply the initialization of state matrix (A) with all zeros. In the absorbing phase each r-bit wide block of the message is XORed with the current matrix state and 24 rounds of the

compression functions are performed. In the squeezing phase the state matrix is simply truncated to the desired length of hash output.

V. INPUT / OUTPUT INTERFACE

To ensure and control the availability of input data at each rising edge of the clock cycle the Load and Acknowledgment signals are used. The length of the input data to be loaded is a 64-bit word at each rising edge of the clock cycle. Similarly if the hash output is ready then it is indicated by putting hash_valid signal to a high logic and then the 64-bit word of hash value is available at each rising edge of the clock. The control path consists of a Finite State Machine, State register, clock and a counter. The data path consists of an Input register Serial-In Parallel- Out (SIPO), a SHA-3 Core and an Output register Parallel-In Serial-Out (PISO). The 1600-bit state matrix is stored in the 1600-bit register. The 0's are used in the initialization phase to initialize the state matrix with all 0's. In the absorbing phase each r-bit wide block of the input message is *XORed* with the r-bit current state matrix and then concatenation is used to combine r and c to form a new state and store it in the 1600-bit Register. The compression function is implemented using LUT-6 primitives. The round constants (RC) are stored in ROM using a 24x64 bit distributed ROM. Respective round constants are addressed during each round using the round number as the ROM address. In the last phase of squeezing the state matrix is simply truncated to the desired length of the hash output. The implementation of SHA-3 core (compression function) is given below in detail.

VI. IMPLEMENTATION OF THE SHA-3 CORE

LUTs are the basic logic building blocks of an FPGA and are used to implement most logic functions of a design. LUT_6 as shown in Figure 4, is a 6-input, 1-output look-up table (LUT) that can either act as an asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function. The INIT parameter for the FPGA LUT primitive provides the logical value of the LUT and consists of a 64-bit Hexadecimal value.

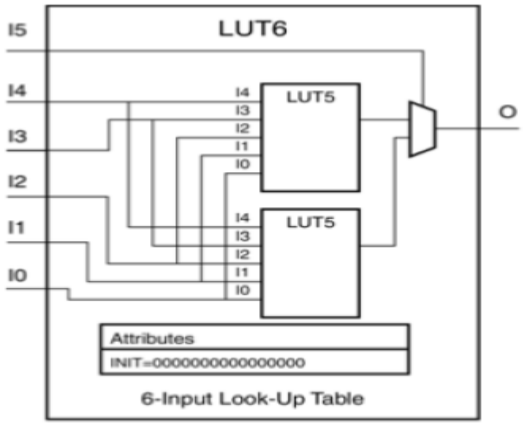

Figure 4. LUT_6 Primitive

It is obvious from the compression function steps that there are two types of logical operations throughout the compression function **(a)** *XOR* logic **(b)** χ (*chi*) logic. In this design we have stored the result of these logics manually in a single LUT_6 primitive as an INIT value. The MSB of the LUT_6 input is used as a control bit for the selection between *XOR* and χ logic. If the control bit is '0' then *XOR* logic is selected and in the case of the control bit being a '1', χ logic is selected. The INIT value of the LUT_6 primitive is 64"h D2D2D2D296696996. This is derived by using the truth table for all possible combinations of 6- input LUTs.

The 1600- bits of the compression function are divided into a 5x5 matrix (A) in such a way that each position of the state matrix contains a 64-bit word. All the bitwise logical operations are performed between these 64-bit words. So, there should be 64 LUTs to perform bitwise logical operation between any 64-bit inputs at a time. Because of this an architecture is proposed using 64 LUT-6 primitives with the same INIT value as shown in Fig. 5. In Fig.5 all the input bits are arranged in a way so that this architecture can be used to perform bitwise logical operation of either *XOR* or *chi* logic between any five 64-bit inputs. As the maximum number of inputs in eq. (1) to eq.(6) are five that's why the Fig.5 architecture is suitable for the implementation of these equations and the remaining input can be used as 'control' bit to select between *XOR* and *chi* logic.

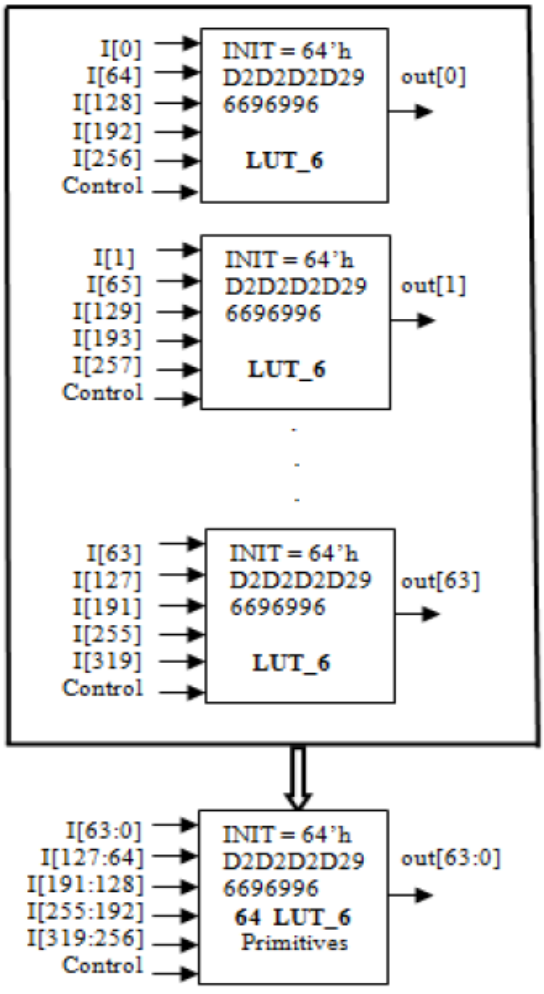

Figure 5. 64 LUT_6 architecture to implement SHA-3 Core

In eq. (1) a 5-input *XOR* function is performed between the 64-bit word of each row of state matrix (A). To implement eq. (1) the architecture of Fig.5 is instantiated 5 times with control bit equal to '0' to select the *XOR* logic. The output of this *XOR* operation is stored in an intermediate register of C[x].

In eq. (2) C[x] position is rotated 1-time in clock wise (C[x+1]) and anti-clockwise (C[x-1]) direction. The 64-bits of C[x+1] are also rotated one time. This rotation operation is implemented manually using concatenation operation. After this rotation operation, the rotated 64-bits of C[x+1] and 64-bits of C[x-1] are *XORed*. To perform this *XOR* operation, again Fig.5 architecture is instantiated 5 times with control bit set to '0' and the output is stored in an intermediate register D[x]. This time only two inputs of Fig. 5 are used (excluding control bit) and the other inputs are grounded.

In eq. (3) the resultant D[x] is *XORed* with each 64-bit row of input state matrix (A) and the entire state of 1600-bits is updated. There are 25 rows of 64-bits in state matrix (A). So, to implement this *XOR* function, Fig. 5 architecture is instantiated 25 times with control bit set to '0'.

Eq. (4) involves the bit rotation of each 64-bit row of the updated 1600-bit state of A[x, y] according to the bit rotation scheme of r[x, y]. After the rotation operation all the rotated bits are stored at new position. The rotation operation is performed by using concatenation.

In eq. (5) chi logic is used that consists of *XOR*, *NOT*& *AND* logical operation. To implement this logic Fig. 5 architecture is instantiated 25 times and entire state of 1600-bit is updated. But now control bit is set to '1' to select *chi* logic.

In eq. (6) only 64-bit of updated A[0, 0] are updated by *XORring* with 64-bit round constant (RC). To implement eq. (6), Fig.5 architecture is instantiated one time with control logic set to '0'.

So, in this way one round of compression function is implemented within one clock cycle. The remaining 23 rounds of the compression functions are completed in the same way sequentially. Therefore, total 24 clock cycles are required for the complete implementation of the compression function.

VII. POWER ANALYSIS USING XILINX XPOWER

Xilinx Xpower [10] is a tool providing the power estimation for an FPGA design. The power of a design can only be analyzed after the process of "place and route". Xpower calculates the power based on the switching activity of the transistors. Any component in an FPGA design with switching activity has a corresponding capacitance model used for the power calculation. There are two methods for power estimation using Xpower: **(1)** A rough estimation by setting an expected toggle rate; **(2)** A more accurate estimation by providing detailed transistor switching activity. Here, we choose the second method to get an accurate result for our designs.

The flow for the power estimation is shown in Fig. 6, where the value change dump file (VCD) is a file to record the signal state at different timeslots and can be obtained by simulation. The switching activity of the circuit can be recorded in the VCD file when we simulate the gate-level netlist of the design. Another purpose of the simulation at gate level is to examine whether the register transfer level (RTL) code is successfully synthesized into the gate level netlist. Finally, Xpower reads the VCD file and generates the power estimation report.

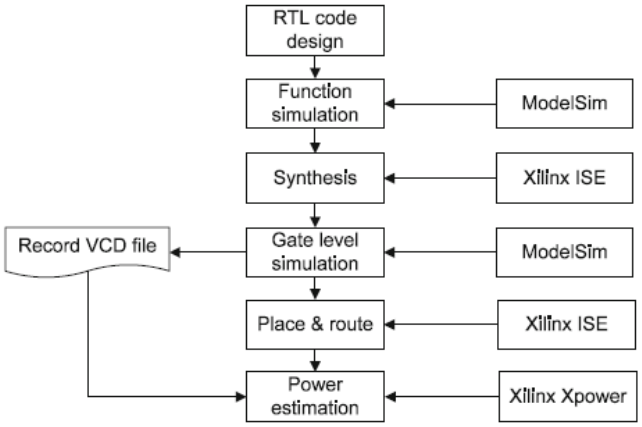

Figure 6. Power estimation flow by Xpower

VIII. IMPLEMENTATION RESULTS

Table II shows the implementation results in terms of area utilization, frequency of the implemented design, power consumption, throughput (TP) and TPA (throughput/area). We have implemented the proposed architecture in all Xilinx FPGA platforms to show which platform is suitable for use in the IoT application environment.

The implementation results from table II shows that the Artix-7 FPGA utilizes less power as compare to the other Xilinx FPGAs and also consumes less area. Virtex FPGAs are power hungry, so these FPGAs are not suitable for IoT applications. The implementation on Spartan-6 consumes less power as compare to Virtex-6 but the throughput is less as compare to other Xilinx FPGAs. The throughput result on Kintex FPGA is better but it utilizes more area making its TPA less attractive.

The throughput and TPA are calculated by using eq. (7) and eq. (8) respectively.

$$TP = Block\ Size / (T.\ N_{CLK}) \quad \textbf{(7)}$$

$$TPA = TP / Area \quad \textbf{(8)}$$

The Block Size is the block size of the message in bits i.e 1088 for 256-bit variant. *T* is the time period of the system clock and N_{clk} is the number of clock cycles required for a valid hash output.

TABLE II. IMPLEMENTATION RESULTS ON XILINX FPGAS

Device	Area (Slices)	Freq. (MHz)	Power (mW)	TP (Gbps)	TPA [TP(Mbps) /Area]
Artix-7	982	192.75	612	8.738	8.89
Virtex-6	1,048	194.78	2026	8.830	8.45
Spartan-6	1,162	111.73	823	5.065	4.35
Kintex-7	1,185	213.17	629	9.660	8.15

IX. CONCLUSION

This work presents an implementation of the newly selected cryptographic hash algorithm called SHA-3 on Xilinx FPGAs. On the basis of the implementation results, the Artix-7 FPGA is recommended for the SHA-3 implementation as this platform is more suitable for IoT applications because of its lower power consumption and it also utilizes less area than the other devices. This implementation proposes a hardware architecture that is based on LUT-6 primitives. The logical functions of SHA-3 core are stored in LUT-6 primitives and these primitives are instantiated for the complete implementation of SHA-3. The power analysis of the implemented design is also provided by using the Xilinx Xpower tool. The authors believe that this implementation is suitable for high data throughput IoT applications that require data integrity and source authentication security services while still remaining power efficient.

REFERENCES

[1] Xiaoyun Wang, X.L., Feng, D., Yu, H.: Collisions for hash functions MD4, MD5,HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, pp. 1–4 (2004), http: // eprint.iacr.org/2004/ 199

[2] Szydlo, M.: SHA-1 collisions can be found in 2^{63} operations. Crypto BytesTechnical Newsletter (2005)

[3] Stevens, M.: Fast collision attack on MD5. ePrint-2006-104, pp. 1–13 (2006), http: //eprint.iacr.org/2006/104.pdf

[4] Federal Register / Vol. 72, No. 212 / Friday, November 2 (2007), Notices, http: // csrc.nist.gov/groups/ST/hash/documents/FR_Notice_ Nov07.pdf

[5] National Institute of Standards and Technology (NIST).SHA-3 Winner announcement, http : // w w w. nis .gov / itl / csd / sha-100212.cfm

[6] F. Henriquez, N. Saqib, D. Prez, and C. Kaya Koc; "Cryptographic Algorithms on Reconfigurable Hardware" Springer, November 2006.

[7] Kuon, I.; Tessier, R.; Rose, J. FPGA Architecture: Survey and Challenges. Found. Trends Electron. Des. Autom. 2007, 2, 135–253.

[8] Xilinx 7 Series Overview; Datasheet DS180; Xilinx. Available online: http: // w w w. xilinx. Com / support / documentation / datasheets / ds1807 Series Overview.pdf.

[9] G. Bertoni, J. Daemen, M. Peeters, G. Assche "The Keccak SHA-3 Submission version 3" pp. 1-14, (2011), http : // Keccak . noekeon . org / Keccak – reference -3.0.pdf

[10] "XPower Tutorial FPGA Design", [online]. Available at: ftp.xilinx. com/pub/documentation/tutorials/xpowerfpgatutorial.pdf