2009 International Conference on Reconfigurable Computing and FPGAs

Lightweight Cryptography for FPGAs

Panasayya Yalla, Jens-Peter Kaps
Department of ECE, Volgenau School of IT&E
George Mason University
Fairfax, VA, USA
Email: pyalla, jkaps@gmu.edu

***Abstract*—The advent of new low-power Field Programmable Gate Arrays (FPGA) for battery powered devices opens a host of new applications to FPGAs. In order to provide security on resource constrained devices lightweight cryptographic algorithms have been developed. However, there has not been much research on porting these algorithms to FPGAs. In this paper we propose lightweight cryptography for FPGAs by introducing block cipher independent optimization techniques for Xilinx Spartan3 FPGAs and applying them to the lightweight cryptographic algorithms HIGHT and Present. Our implementations are the first reported of these block ciphers on FPGAs. Furthermore, they are the smallest block cipher implementations on FPGAs using only 117 and 91 slices respectively, which makes them comparable in size to stream cipher implementations. Both are less than half the size of the AES implementation by Chodowiec and Gaj without using block RAMs. Present's throughput over area ratio of 240 Kbps/slice is similar to that of AES, however, HIGHT outperforms them by far with 720 Kbps/slice.**

***Keywords*-lightweight cryptography; HIGHT; Present; FPGA;**

I. Introduction

Ubiquitous computing represent the third era of computing devices after mainframes and personal computer for first and second eras. Radio frequency identification (RFID) tags and wireless sensor network (WSN) nodes are a few examples which are being used for automated electronic toll systems, identification tags for food products, pets, clothing and so on. This brings us close to the threshold of pervasive computing. The mass deployment of these device brings serious concerns for security and privacy. The traditional cryptographic algorithms may not be suitable for these device as they have limited memory and computational power along with serious power constraints. This led to development of new branch of cryptography called lightweight cryptography [1]. HIGHT [2] and Present [3] were developed specifically for lightweight cryptography. AES and Camellia, though not considered lightweight, are also being used on these devices.

Until now, lightweight cryptography is targeted towards application specific integrated circuits (ASICs). ASICs involve high non-recurring engineering cost and long time to market where as Field Programmable Gate Arrays (FPGAs) involve low non-recurring engineering cost and less time to market. The dominant factor favorable to ASICs is their lower power consumption, which is of primary concern for lightweight cryptographic devices and their lower cost in large volumes. With the advent of low-cost and low-power FPGAs [4], we expect them to become popular for battery powered applications such as WSN nodes. Hence, they are a targeted for lightweight cryptographic applications. Reconfigurability of FPGAs allows the system to be upgraded if ever the need arises which is not possible with ASICs. Further more, lightweight crypto implementations lead to area saving over traditional implementations. This enables a designer to add crypto to an existing design at a minimal cost or to reduce the overall area consumption which might lead to cost saving as the design might now fit into a smaller, cheaper FPGAs. We designed lightweight architectures of Present and HIGHT for Xilinx Spartan-3 FPGAs. The ciphers considered are of full strength security i.e 128-bit key length, even though traditional lightweight cryptography considers 80-bit key length to be sufficiently secure.

II. Lightweight Ciphers

A. HIGHT

The block cipher HIGHT [2] was developed by a group from Korea University, National Security Research Institute (NSRI) and Korea Information Security Agency (KISA) in 2006. HIGHT (HIGH security and light weighT) is a 64-bit block cipher with 128-bit key length. It uses generalized Feistel structure with 32-rounds containing simple operations such as XOR, modular addition in the group of 2^8 elements, and bitwise rotation. The absence of traditional substitution layer, its Feistel structure and byte oriented operations make it suitable for low-cost, low-power and lightweight implementations. The original design presented in [2] was implemented on ASICs with 3048 gates. The HIGHT algorithm was modified [5] to reduce the critical path in the key scheduler which also reduced the area to 2608 gates. The initial security analysis presented in [2] showed that HIGHT only 19 rounds is secure. Subsequent analysis [6] increased the rounds required to 28.

B. Present

Present is an ultra-lightweight block cipher proposed by A. Bogdanov, L.R. Knuden and G. Leander et al. [3].

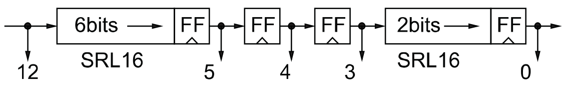

Figure 1. 12-bit Shift Register

Present is a 31-round Substitution-Permutation (SP) network with a block size of 64-bit and 80-bit or 128-bit key lengths. In this paper a 128-bit key length is considered. Present was designed by incorporating some features of Serpent [7] and Data Encryption Standard (DES) [8] which demonstrated excellent performance in hardware. The non-linear substitution layer, i.e. S-box in Present is similar to that of Serpent and the linear permutation layer to that of DES. The original Present proposal provides a basic security analysis [3]. Further-more, analysis was performed in [9], [6] and [10] showing that 31-round Present with 31-rounds is considered secure.

III. Optimization Techniques

Designing compact architectures in FPGAs depends on effective use of architectural features provided in the targeted FPGAs. Xilinx Spartan-3 FPGAs have features such as Look Up Table (LUT) based 16-bit shift register (SRL16) and Distributed Random Access Memory (DRAM) which can be employed to improve the performance and decrease the area of a given design by an order of magnitude. Our techniques can be adapted to similar features provided by other FPGA vendors.

A. Xilinx Spartan-3 FPGA Architecture

1) FPGA Structure: The fundamental logic unit in Xilinx FPGAs is a slice, which contains mainly two four input LUTs and two flipflops. Half the slices of a chip are called SLICEMs. Their LUTs can be configured as 16-bit shift registers (SRL16) or as 16-bit distributed RAMs called DRAMs.

2) Shift Register (SRL16): The number of slices required for implementing a shift register depends on the number of bits to be stored and the number of taps. Taps are positions of a shift register where data can be written to or read from. Each tap is configured as a flipflop. Fig. 1 shows an example of a 12-bit shift register with taps at bits 5, 4, 3, and 0. The synthesis tool infers SRL16s when a shift register is described in hardware descriptive language (HDL) without reset. Otherwise it builds a shift register from flipflops.

3) Distributed RAM (DRAM): DRAMs offer fast and localized memory. They can be cascaded for realizing deeper memories with minimal penalty on timing. Distributed RAM supports two types of memories: single-port RAM and dual-port RAM. Both have synchronous write and either synchronous or asynchronous read. The area depends on the number of output bits and memory depth. Depending on the specific logic synthesis tool used, DRAMs can be instantiated directly or inferred based on the hardware description of RAMs with no reset.

B. Plaintext and Key Storage

The most area consuming components of cryptographic algorithms are data and key storage. DRAM and shift register are two options for efficient memory implementation. In order to develop lightweight architectures, the algorithm implementations are scaled down to use either an 8-bit or a 16-bit datapath. Key and data are loaded either 8-bits or 16-bits depending on the implementation. Loading into shift register is simpler as the number of control bits needed are less as compared to DRAM which needs addressing. The size of the address increases with the number of words to be stored in DRAM. On the other hand, the control bits of shift register are independent of size of the data it stores. However, some ciphers require intermediate data values. For example in the case of Camellia [11], multiple values from disparate locations are required to perform a single computation. This makes use of DRAM more appropriate. In our cipher implementations of Present, plaintext is stored in a shift register as data is needed in a regular order. In case of HIGHT, plaintext is stored in DRAM due to its generalized feistel structure. The key scheduling in ciphers Camellia, HIGHT and Present involve shifting of key which make use of shift register more apt. However, the key in HIGHT is need in a different order during initial and final transformations compared to round operations which is difficult to accomplish with a shift register. In this case, a DRAM is used to store the key.

C. Control Logic

Finite State Machines (FSM) are used for realizing the control logic of complex systems. Traditionally, FSMs are implemented using flipflops and combinational logic. However, this type of FSM implementation is complex and not efficient. The use of RAM blocks for sequential logic led to ROM-based FSM implementations which have been shown to be efficient [12], [13], [14]. The control signals for each operation are combined to one control word. These control words are stored in a memory location which can be accessed by an address. DROMs are inferred by holding the write signal low for DRAMs. ROM- based FSMs have additional advantages. The maximum frequency at which a ROM-based FSM operate is independent of the complexity of the circuit. This method is also proved to save power [15]. For our HIGHT and Present architectures, the control signals are generated by a counter and some additional logic. The size of the control word is reduced by removing any control signals which repeat a short sequence of values many times, such as control signals for round operations, from the main controller and assigning them to a sub-controller. We use this technique for camellia where the control signals for round function f are generated by a sub-controller called F-controller.

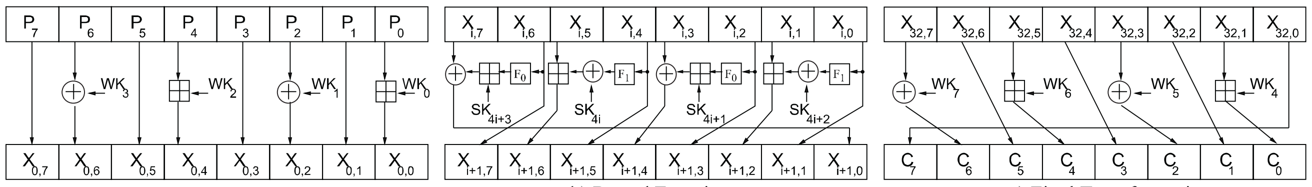

Figure 2. HIGHT

IV. HIGHT

A. Algorithm

The HIGHT algorithm consists of 32-rounds with initial and final transformations before the first and after the last rounds respectively. The plaintext P and ciphertext C are split into eight 8-bit blocks $P_7, \cdots, P_0$ and $C_7, \cdots, C_0$ and the original key K into sixteen 8-bit blocks $K_{15}, \cdots, K_0$.

The initial transformation uses the four whitening key bytes WK_0, WK_1, WK_2, and WK_3 to transform a plain text P into the input of first round function, $X_0 = X_{0,7} \parallel \cdots \parallel X_{0,0}$. In the final transformation, the data is shifted towards right and transforms $X_{32} = X_{32,7} \parallel \cdots \parallel X_{32,0}$ into cipher text C by with the four whitening keys WK_4, WK_5, WK_6 and WK_7. Both transformations performs a XOR or modular addition as shown in Fig. 2a) and Fig. 2c). The eight 8-bit whitening keys for initial and final transformation $WK_7, \cdots, WK_0$ are generated using Eq. (1).

$$\text{For } 0 \leq l, m \leq 7$$

$$WK_l \leftarrow K_{12+l},\ WK_{l+4} \leftarrow K_l \qquad (1)$$

$$SK_{16*l+m} \leftarrow K_{(m-l) \bmod 8} \boxplus \delta_{16*l+m} \qquad (2)$$

$$SK_{16*l+m+8} \leftarrow K_{((m-l) \bmod 8)+8} \boxplus \delta_{16*l+m+8} \qquad (3)$$

The round function uses two auxiliary functions F_0 and F_1 described in Eq. (4) and (5) respectively, along with XOR and modular addition operations. The two functions F_0 and F_1 provide bitwise diffusion which is similar to linear transformation from $GF(2)^8$ to $GF(2)^8$. The round function transforms the input $X_i = X_{i,7} \parallel \cdots \parallel X_{i,0}$ into $X_{i+1} = X_{i+1,7} \parallel \cdots \parallel X_{i+1,0}$ for $i = 0, \cdots, 31$, which is shown in Fig. 2b).

The i^{th} round key RK_i consists of four 8-bit subkeys $SK_{4*i}, SK_{4*i+1}, SK_{4*i+2}$, and SK_{4*i+3} which are generated through Eq. (2) and (3). The 7-bit constants $\delta_0, \cdots, \delta_{127}$ are generated by the 7-bit LFSR h. The characteristic polynomial of h is x^7+x^3+1 in $\mathbb{Z}_2$ with a period of 2^7-1=127. The initial value of h is set to 1011010_2.

$$F_0(x) = x^{\lll 1} \oplus x^{\lll 2} \oplus x^{\lll 7} \qquad (4)$$

$$F_1(x) = x^{\lll 3} \oplus x^{\lll 4} \oplus x^{\lll 6} \qquad (5)$$

B. Lightweight Architecture of HIGHT

We reduce the area consumption for HIGHT by scaling the 64-bit algorithm to 8-bit and apply optimizations described in Section III. This reduction does not come at the cost of temporary storage or multiplexers.

1) Data Storage : The round function involves shifting which suggest that a shift register is the most efficient solution for data storage. However, the generalized Feistel structure of HIGHT leads to a misalignment of data when a shift register is used. Realignment requires additional clock cycles. This reduces the throughput and the more complex logic increases the area consumption. Therefore, we use a DRAM which is more efficient both in terms of area and latency. A dual-port DRAM is used as the round function requires two 8-bit blocks of data for computing one 8-bit block. The shifting involved in round function is accomplished by addressing. The addresses needed for all operations are generated by using three 3-bit multiplexers M6, M7, and M8, 2-bit and 3-bit adders, 12-bit shift register SR and a 3-bit counter C3 shown in Fig 3. The address from M8 is used for both reading and writing while M7 is used only for reading. The control signals for all operations are generated by using a 5-bit counter and some logic functions.

2) Round Function, Initial and Final Transformations: The addresses for round function are generated by shift register (SR) and a 3-bit adder. The initial values of SR required for first round are computed by using 2 LSB bits of C3. The addresses for the next round are generated by loading the result of 3-bit adder into SR. This way of generating the addresses reduces the complexity of the control logic and avoids extra clock cycles needed for shifting of data. Each round operation requires 4 clock cycles. The initial and the final transformations are performed by using the datapath of the round function with use of two extra multiplexers M2 and M3. The addresses for both transformations are generated by C3. The initial transformation is performed while the data is being loaded into the shift register which save clock cycles.

3) Key Storage and Scheduling: The 128-bit key is stored in a single-port DRAM. The subkeys and whitening keys are generated by using two 3-bit counter C1 and C2 and a 2x1 multiplexer M5. The MSB bit of the key address which is also used as selection bit for M5 is generated from the output of the 5-bit counter used in control logic. The two 3-bit counters are used for addressing instead of one 4-bit counter to accomplish shifting involved in key scheduling.

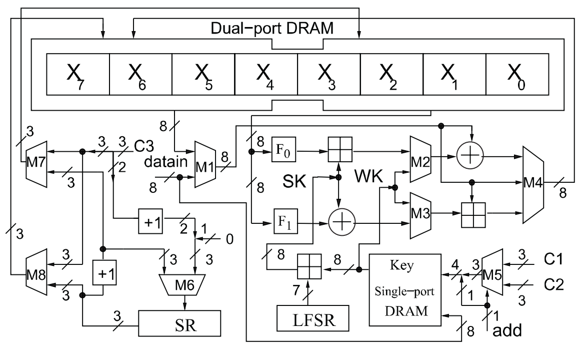

Figure 3. Top level block diagram of HIGHT

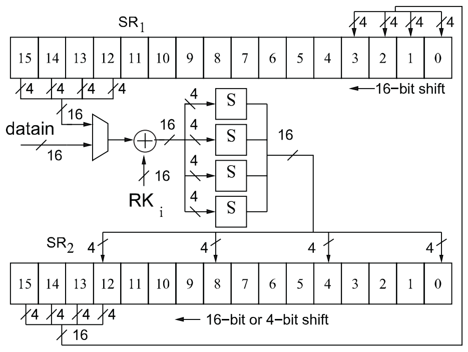

Figure 4. 16-bit datapath of Present

V. Present

A. Algorithm

The initial state of Present $b_{63} \cdots b_0$ is defined by the 64-bit plaintext P. The round function changes this state in every iteration i for $0 \leq i \leq 15$ through its three stages: addRoundkey, sBoxLayer and pLayer. The addRoundkey operation XORs the current state $b_{i,63} \cdots b_{i,0}$ with the i^{th} round key $RK_i = rk_{i,63} \cdots rk_{i,0}$ as follows:

$$\text{For } 0 \leq j \leq 63: \quad b_{i,j} \leftarrow b_{i,j} \oplus rk_{i,j}.$$

The second stage is the non-linear sBoxLayer which consists of 16 copies of a 4-bit to 4-bit S-box. Each S-box is applied to S(w_i) for $w_{15}, \cdots, w_0$ where $w_i = b_{4*i+3} \parallel b_{4*i+2} \parallel b_{4*i+1} \parallel b_{4*i}$ for $0 \leq i \leq 15$. The linear bit permutation pLayer is the third stage of the round operation which can be described by Eq. (6).

$$\begin{gathered} \text{For } 0 \leq i \leq 15 \\ b_i \leftarrow b_{4*i}, \quad b_{i+16} \leftarrow b_{4*i+1}, \\ b_{i+32} \leftarrow b_{4*i+2}, \quad b_{i+48} \leftarrow b_{4*i+3}. \end{gathered} \tag{6}$$

The current round key RK_i are the 64 most significant bits (MSB) of the current state of the key register K. The round key RK_{i+1} for the next round $i+1$ is generated by shifting the key register $K = k_{127}k_{126} \cdots k_1k_0$ to the left by 61 bits and passing the left most 8-bits through two S-boxes of Present. The 5-bits $k_{66}k_{65}k_{64}k_{63}$ are XORed with the 5-bit round counter. The resultant 64 MSB of K form the round key bits $RK_{i+1} = k_{127} \cdots k_{64}$.

B. Lightweight Architecture of Present

We achieved area reduction by scaling the 64-bit implementation to a 16-bit implementation and by applying the optimization techniques stated in Section III for several components used in this implementation. Scaling the implementation to 8-bit would decrease the throughput drastically and yield a very small area reduction due to the complexity of the permutation operation. Our implementations of wider datapaths led to a significant increase in area consumption. The top level datapath is shown in Fig. 4.

1) Data storage: The state $b_{63} \cdots b_0$ is stored in the shift register SR_1 for the reason specified in III. It performs a 16-bit circular left-shift per clock cycle. We consider the 64-bit shift register as a combination of sixteen 4-bit block 15,$\cdots$,0. The 16 MSB are tapped out of SR_1 for the round operation.

2) S-box Implementation and Permutation Layer: The round operation starts by XORing the incoming data with the round key RK_i and applying the result to four S-boxes. Present's 4-bit to 4-bit S-boxes are implemented in a single LUT each. Our architecture uses four S-boxes for round operations and two for key scheduling. The Permutation function is implemented by using the shift register SR_2 which performs a shift by 4 bits during round operation and by 16 bits after each round when copying its content into SR_1. Furthermore, the 16-bit output from the S-boxes is given as input to the blocks 12, 8, 4, and 0 of SR_2. During first clock cycle of the round operation the 4-bits blocks 15, 11, 7 and 3 are computed from the 16 MSB of SR_1 and placed in position 12, 8, 4, and 0 of SR_2. In the subsequent clock cycle SR_2 is shifted by 4 bits and blocks 14, 10, 6 and 2 are computed and placed in the now empty positions 12, 8, 4, and 0 of SR_2. These operations repeat for another two clock cycles to complete the round function. This results in a total of 8 clock cycles for each round operation.

3) Key Storage and Scheduling: The key is stored in a 128-bit shift register which performs a 16-bit circular left-shift. The first round key RK_1 is obtained during the first four clock cycles by tapping the 16 MSB from the key and passing them to the *RKGen* function. However, during these four clock cycles the key was shifted by 64-bit. Subsequent round keys require only a shift by 61 bits which is not possible with a 16-bit shifts. We overcome this problem by placing three extra taps on the shift register and using two 3-bit registers A and B along with several multiplexers (see Fig. 5). The 3 bits that were "lost" during generation of the first round key are stored in register A. For subsequent round keys, the value from register A and the 13 MSB from the key are passed to *RKGen*. However, when generating the round key RK_2 we are facing the same problem of

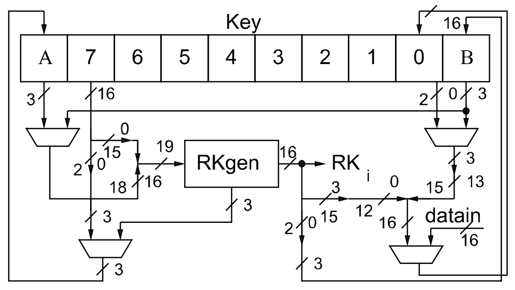

Figure 5. Key scheduling of Present

shifting by 64-bit again and compensate by storing three bits in register B. From now on, *RKgen* gets the value from registers B||A||11 MSB from key. When we write the round keys back into the shift register we take the 3-bit difference into account. Therefore, no additional registers are required for subsequent rounds. The *RKgen* function consists of two S-boxes for S-box operation, a 5-bit XOR to compute the XOR with the round counter, and multiplexers to choose the appropriate bits for round key generation. The output of the *RKgen* function is the round key.

VI. Implementation Results

Our designs of HIGHT and Present were described in VHDL, synthesized for the Xilinx Spartan-3 FPGA-XC3S50-5 device using Xilinx ISE 9.2i and simulated with Active HDL 7.2. All results are after place and route. All our cipher implementations operate in electronic code book mode (ECB), are not pipelined, and generate all round keys on the fly without requiring additional clock cycles.

Table I shows the detailed results of our implementations of HIGHT and Present. We also included the implementation results of Camellia and TinyXTEA which were done by our group previously [11], [16]. The results for AES were obtained by using the VHDL code for the ASIC implementation reported in [17] and synthesizing it for our target FPGA. We like to note that this implementation is not optimized for FPGAs. Camellia and AES encrypt blocks of 128-bit data, whereas the other algorithms operate on 64-bit data blocks. Therefore, AES and Camellia implementations require more storage. Furthermore, AES and Camellia have 8x8 S-boxes which occupy 64 slices or 16% and 20% respectively of the total design area in our implementations. Present's 4x4 S-box occupies only 2 slices. Our implementation of Present uses six S-boxes which occupy 10% of the total design area. AES and TinyXTEA-3 use registers (i.e. flipflops) for data and key storage. Even though the total number of flipflops needed is far smaller than the number of LUTs used, the addressing logic contributes to the area consumption. The Camellia implementation uses 88 SRL-16 elements, which would be capable of storing a maximum of 1,408 bits, to store its two 128-bit keys. Unfortunately, the round key generation shifts the key by 15 and 17 bits. This irregular shift requires many additional tapings causing the high number of SRL-16 elements. Implementing shifts in multiples of 8 require less area. The same can be observed in Present's key schedule as its involves 61-bit shifts. HIGHT makes extensive use of DRAM elements for both, data and key storage and uses SRL-16s in its control logic. Present uses SRL-16s for both. DRAM and SRL-16 elements are an ideal choice for storing data and key provided that the algorithm is regular which leads to a simple control logic. Camellia is an example for an algorithm with high irregularity, therefore DRAM and SRL-16 elements cannot be used to full effect. Implementing permutation functions that span more than 8 or 16 bits also increases the area consumption and latency for lightweight implementations in FPGAs.

Table I
Lightweight Implementation Results for Xilinx xc3s50-5

Block Cipher	Flipflops	LUTs	DRAMs	SRL-16s	Slices	Implementation Choices for	
						Data	Key
Present	114	159	0	16	117	SRL	SRL
HIGHT	25	132	24	3	91	DRAM	DRAM
Camellia [11]	164	420	16	88	318	DRAM	SRL
TinyXTEA-3 [16]	226	424	0	0	254	FF	FF
AES [17]	338	531	0	0	393	FF	FF

Table II compares our implementations with Camellia, TinyXTEA-3, AES and the eSTREAM portfolio ciphers [21]. The stream ciphers outperform all block cipher implementations with respect to the throughput/area metric. However, they are defined for 80-bit keys and only MICKEY and Grain offer 128-bit versions. Stream ciphers are still considered immature [21] and only recently the stream cipher F-FCSR-H was removed from the portfolio. AES [19] has the highest throughput of the block ciphers followed by HIGHT. However, HIGHT has a better throughput to area ratio and consumes only half the size and no block rams. For the throughput to area ratio computation we added 300 slices to the area of AES [19] and 140 to AES [18] to compensate for the block ram usage [18]. We realize that AES [19] was implemented on a Spartan II device which can not be compared directly with a Spartan-3 implementation but we doubt that this alone would cause a factor three difference in delay.

VII. Conclusion

Lightweight implementations of cryptographic algorithms for FPGAs is going to become an important research area due to the introduction of FPGAs for battery powered devices. In this paper we introduced the first lightweight implementations of the block ciphers HIGHT and Present on FPGAs. Our implementation of HIGHT consumes less than 100 slices, encrypts data at 65 Mbps and has a better throughput over area ratio than the previously published lightweight implementation of AES [19]. Furthermore, we introduced optimization techniques for lightweight implementations

Table II
RESULTS FOR PRESENT AND HIGHT COMPARED TO OTHER BLOCK CIPHERS AND THE eSTREAM PORTFOLIO CIPHERS ON FPGA

Design	Maximum Delay (ns)	Clock Cycles per block	Block Size (bits)	Key Size (bits)	Area (slices)	Block RAMs	Throughput (Mbps) at f_{max}	Throughput/ Area (Mbps/slice)	Device
Present	8.78	256	64	128	117	0	28.46	0.24	xc3s50-5
HIGHT	6.12	160	64	128	91	0	65.48	0.72	xc3s50-5
Camellia [11]	7.95	875	128	128	318	0	18.41	0.06	xc3s50-5
AES [17]	14.21	534	128	128	393	0	16.86	0.04	xc3s50-5
AES 8-bit [18]	14.93	3900	128	128	124	2	2.2	0.01	xc2s15-6
AES [19]	20.00	46	128	128	222	3	139	0.27	xc2s30-5
TinyXTEA-3 [16]	15.97	112	64	128	254	0	35.78	0.14	xc3s50-5
Grain v1 [20]	5.10	1	1	80	44	0	196	4.45	xc3s50-5
Grain 128 [20]	5.10	1	1	128	50	0	196	3.92	xc3s50-5
MICKEY v2 [20]	4.29	1	1	80	115	0	233	2.03	xc3s50-5
MICKEY 128 [20]	4.48	1	1	128	176	0	223	1.27	xc3s50-5
Trivium [20]	4.17	1	1	80	50	0	240	4.80	xc3s50-5
Trivium (x64) [20]	4.74	1	64	80	344	0	13,504	39.26	xc3s400-5

that can also be applied to other algorithms. Investigating the robustness of lightweight implementations against side channel analysis and implementing lightweight asymmetric cryptosystems is future work.

REFERENCES

[1] J.-P. Kaps, G. Gaubatz, and B. Sunar, "Cryptography on a speck of dust," *Computer*, vol. 40, no. 2, pp. 38–44, Feb 2007.

[2] D. Hong *et al.*, "HIGHT: A new block cipher suitable for low-resource device," in *CHES 2006*, ser. LNCS, L. Goubin and M. Matsui, Eds., vol. 4249. Springer, 2006, pp. 46–59.

[3] A. Bogdanov *et al.*, "PRESENT: An ultra-lightweight block cipher," in *CHES 2007*, ser. LNCS, vol. 4727. Springer, 2007, pp. 450–466.

[4] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger, "A 90nm low-power FPGA for battery-powered applications," in *FPGA '06*. New York, NY, USA: ACM, 2006, pp. 3–11.

[5] L. Young-Il, L. Je-Hoon, Y. Younggap, and C. Kyoung-Rok, "Implementation of HIGHT cryptic circuit for RFID tag," *IEICE Electronics Express*, vol. 6, no. 4, pp. 180–186, 2009.

[6] O. Ozen *et al.*, "Lightweight block cipher revisisted: Cryptanalysis of reduced round PRESENT and HIGHT," in *ACISP*, ser. LNCS, vol. 5594. Springer, 2009, pp. 90–107.

[7] E. Biham, R. Anderson, and L. Knudsen, "Serpent: A new block cipher proposal," in *FSE 1998*, ser. LNCS, vol. 1372. Springer, January 1998, pp. 222–223.

[8] *Data Encryption Standard (DES)*, National Institute of Standards and Technology, FIPS Publication 46-3, Oct 1999.

[9] M. Wang, "Differential cryptanalysis of reduced-round PRESENT," in *AFRICACRYPT 2008*, ser. LNCS, S. Vaudenay, Ed., vol. 5023. Springer, 2008, pp. 40–49.

[10] B. Collard and F.-X. Standaert, "A statistical saturation attack against the block cipher PRESENT," in *CT-RSA*, ser. LNCS, vol. 5473. Springer, 2009, pp. 195–210.

[11] P. Yalla and J.-P. Kaps, "Compact FPGA implementation of Camellia," in *FPL 2009*, M. Daněk, J. Kadlec, and B. Nelson, Eds. IEEE, Aug. 2009, pp. 658–661.

[12] M. Rawski, H. Selvaraj, and T. Luba, "An application of functional decomposition in ROM-based FSM implementation in FPGA devices," *J. Syst. Archit.*, vol. 51, no. 6-7, pp. 424–434, 2005.

[13] V. Skylarov, "Synthesis and implementation of RAM-based finite state machines in FPGAs," in *FPL '00*, ser. LNCS, R. W. Hartenstein and H. Grünbacher, Eds., vol. 1896. Springer-Verlag, 2000, pp. 718–728.

[14] I. García-Vargas *et al.*, "Rom-based finite state machine implementation in low cost FPGAs," in *ISIE*. IEEE, June 2007, pp. 2342–2347.

[15] A. Tiwari and K. A. Tomko, "Saving power by mapping finite-state machines into embedded memory blocks in FPGAs," in *DATE '04*. IEEE Computer Society, 2004, p. 20916.

[16] J.-P. Kaps, "Chai-tea, cryptographic hardware implementations of xTEA," in *INDOCRYPT 2008*, ser. LNCS, D. Chowdhury, V. Rijmen, and A. Das, Eds., vol. 5365. Springer, Dec 2008, pp. 363–375.

[17] J.-P. Kaps and B. Sunar, "Energy comparison of AES and SHA-1 for ubiquitous computing," in *EUC-06*, ser. LNCS, vol. 4097. Springer, Aug 2006, pp. 372–381.

[18] T. Good and M. Benaissa, "AES on FPGA from the fastest to the smallest." in *CHES 2005*, ser. LNCS, J. R. Rao and B. Sunar, Eds., vol. 3659. Springer, 2005, pp. 427–440.

[19] P. Chodowiec and K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *CHES 2003*, ser. LNCS, vol. 2779. Springer, Sep. 2003, pp. 319–333.

[20] D. Hwang *et al.*, "Comparison of FPGA-targeted hardware implementations of eSTREAM stream cipher candidates," in *SASC Workshop 2008*, Feb 2008, pp. 151–162.

[21] C. Cid and M. Robshaw, "The eSTREAM portfolio 2009 annual update," eSTREAM, ECRYPT Stream Cipher Project, Tech. Rep., Jul 2009.