RAM-Based Ultra-Lightweight FPGA Implementation of PRESENT

Elif Bilge Kavun and Tolga Yalcin
Chair of Embedded Security
Horst Görtz Institute, Ruhr-University Bochum
Bochum, Germany
Email: elif.kavun@rub.de, tolga.yalcin@rub.de

Abstract—In this paper, two different FPGA implementa-
tions of the lightweight cipher PRESENT are proposed. The
main design strategy for both designs is the utilization of
existing RAM blocks in FPGAs for the storage of internal
states, thereby reducing the slice count. In the first design,
S-boxes are realized within the slices, while in the second
design they are also integrated into the same RAM block used
for state storage. Both designs are well suited for lightweight
applications, which are implemented on low-cost FPGA/CPLD
devices. Besides low-area, a reasonable throughput is also
obtained even though it is not the first concern. In addition
to a single block RAM, the two designs occupy only 83 and
85 slices and produce a throughput of 6.03 and 5.13 Kbps at
100 KHz system clock on a Xilinx Spartan XC3S50 device,
respectively.

Keywords—PRESENT; FPGA; lightweight; RAM

I. INTRODUCTION

In recent years, lightweight devices have started to play
an important role in our daily lives. Examples can be RFID
tags or wireless sensor nodes which handle systems like
electronic payment, product identification, tracking, alarm
information, intrusion detection, habitat monitoring [1], [2],
structural monitoring, and emergency medical response [3],
[4]. The large deployment of these lightweight services
resulted in security and privacy problems. In order to make
them secure, special cryptographic techniques should be
applied as they have limited resources and require low-
power consumption. Therefore, lightweight cryptography is
used for securing these constrained devices. This can be
done either in software using low-end embedded processors,
or in hardware via custom ASICs (in case of RFIDs) and
low-cost FPGAs. FPGAs offer the additional advantage of
reconfigurability, which is desired especially in the sensor
nodes. Several new lightweight block ciphers and more
recently hash functions have emerged as a result of con-
tinuing research in this field [5], [6], [7]. PRESENT is
perhaps the best known of the lightweight cryptographic
algorithms, mainly due to its specific design targeted for
ease of implementation both in hardware and software.

Sensor networks have been proposed and deployed for a
wide variety of applications such as . Typical sensor network
applications re-quire low-throughput devices that have low
real-time computation requirements.

Until now, several implementations of lightweight crypt-
tographic algorithms have been performed on application
specific integrated circuits (ASICs). While very desirable
for mass production and low-power consumption, ASICs
suffer from high costs associated with high non-recurring
engineering (NRE) costs and long manufacturing times,
which may be the killing factor for low volume products.
Field programmable gate arrays (FPGAs) present attractive
alternatives for this portion of the market with their low or
zero NRE costs, and short time-to-market properties.

Although ASICs still have lower costs in large volumes,
for customers willing to produce small amounts of sensor
nodes or RFIDs, low-cost and low-power FPGAs seem
to be the ideal solution. With the latest advances in the
FPGA technologies and architectures, they are expected to
be preferable for battery powered applications like wireless
sensor nodes [8], [9], presenting a popular platform for
lightweight applications. Furthermore, the reconfigurable
nature of FPGAs makes them even more attractive in appli-
cations which may require regular hardware updates and/or
modifications.

Most of these applications require one form of security,
either to secure communications between various devices,
or to provide authentication information. The PRESENT
block cipher, whose suitability for compact implementation
on ASICs has been demonstrated several times can also be
used on FPGAs. Until now, register based implementations
of PRESENT on FPGAs have been reported [10], [11].
However, none of them have made use of the already existing
block RAMs on FPGAs.

The block RAMs on FPGAs can be considered as one
of their major advantages over ASICs, which require use
of special memory generators adding further onto the man-
ufacturing costs. However, they can also be considered as
a curse, especially in massively computational applications,
where registers are dominantly used for state data storage
and RAMs are left unutilized. Block ciphers are perfect to
such applications.

In this study, we aim to demonstrate that block RAM
based implementations of block ciphers are also possible
and viable on FPGAs, which result in more slices left for
other applications. We use PRESENT as our target algorithm
mainly due to its lightweight nature and popularity, and
come up with two designs, which, to the best of our knowledge, are the most compact block cipher implementations on FPGAs.

The rest of the paper is organized as follows: Section II provides the previous work. In Section III, the PRESENT algorithm is summarized. In Section IV, the proposed designs and their implementations are described in detail. The performance results and their comparisons with previous works are presented in Section V. Finally, Section VI concludes the paper by also giving future directions.

II. PREVIOUS WORK

Due to the relatively high-cost and high power consumption of FPGAs with respect to ASICs in mass amounts, they have not been the target application platform for lightweight applications until recently. As a result, there has been very few works done on the design and implementation of lightweight ciphers on FPGAs. The two most recent and promising research works are presented in [10] and [11]. The first one is mainly a comparison of the PRESENT cipher against HIGHT on FPGAs. The second one is the detailed survey of PRESENT implementations on FPGAs using different logic optimizers. Both designs rely on registers for state storage. Therefore, they both result in slice counts above 100, while offering high data throughputs. However, we believe that for lightweight applications, lower slice count is more important than high throughput.

III. PRESENT CIPHER

PRESENT is an ultra-lightweight symmetric encryption algorithm published by A. Bogdanov et al. [5]. The algorithm was aimed for lightweight applications such as RFID tags, sensor nodes and internet of things. It is a substitution-permutation network (SPN) with a block size of 64-bit and key size of 80-bit or 128-bit. 64 most significant bits of the supplied key is used as the round key. In this work, 128-bit key length is considered for comparison with previous works. PRESENT has 31 regular rounds and a final round which is only a key addition step. A regular round consists of a key addition step, a substitution layer and a permutation layer. The substitution layer composed of sixteen 4x4-bit S-boxes. S-box is given in Fig. 1. In permutation layer, bit i of the state is moved to bit position P(i). The permutation layer is given in Fig. 2.

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(x)</td>
<td>C</td>
<td>0</td>
<td>5</td>
<td>B</td>
<td>9</td>
<td>0</td>
<td>A</td>
<td>D</td>
<td>3</td>
<td>E</td>
<td>F</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 1. PRESENT S-box.

The key schedule of PRESENT consists of 61-bit left rotation, an S-box (two S-boxes for 128-bit key) and XOR with a round counter. It uses the same S-box for the datapath and key schedule. The key is rotated 61 bit to the left, the left-most 4 bits (8 bits for 128-bit key) are passed through the S-box and the round counter value i is XORed with bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of key with the least significant bit of the round counter on the right.

IV. IMPLEMENTATION

In this work, in order to provide security for lightweight applications, PRESENT algorithm is used as mentioned before. As the targeted platform is FPGA, FPGA-specific properties are considered to obtain a smaller core. Using one of the existing block RAMs for storing state and key data instead of registers is one important technique to get a smaller slice count. Even in this technique, there are two different approaches: One approach is to implement S-boxes on slices, and the second approach is to store S-box in RAM as a lookup table. The second technique is expected to give a smaller slice count, as the S-boxes are not using slices, but existing RAMs instead. The only drawback of this technique is the increase on cycle count and complexity of its control block compared to the first approach.

For both approaches, 64-bit state is represented as 4x4 state matrix with 4-bit entries. In order to store the data in 4x4 matrix, 4x16-bit entries are required for state and 8x16-bit entries are required for key (128-bit key). However, this requirement is doubled by using memory in a ping-pong fashion: Rounds of PRESENT are named as even and odd rounds, and the result of even round is written into odd-round addresses. Likewise, the result of the following odd round is written into even addresses. By doing so, existing memory space is used instead of registers. This scheme is shown in Fig. 3.

A. State Processing

While S-Boxes can be applied to each 16-bit entry in parallel; due to the nature of permutation layer which is shown in Fig. 4, it is not possible to do the same thing for permutation. Therefore, a different approach is applied. This approach can be best explained by a toy version of PRESENT with only 3 rounds as shown in Fig. 3. In the first round (Round-1), the first 16-bit of input data and key are read from address S0 and K0, then XORed and S-Boxed in the following cycle. The result is sent to a temporary register to appear in the next cycle. Also, the first target address S4 is read in the same cycle.

In the next cycle, permutation is performed by using the first bit of every nibble in temporary register (which is the
XORed and S-BOXed value) and stored to S4 by shifting it by 4 bits to left. At the same time, temporary register is shifted left by 1-bit, in order to get the correct permutation values for the next cycle. The same operation is performed for addresses S5, S6 and S7. At the end of 6 cycles, first nibbles of S4, S5, S6 and S7 are ready. The procedure is repeated for all other input addresses S1, S2 and S3. The cycle count for each of these is also 6 cycles. In total, this processing of state takes $4 \times 6 = 24$ cycles for each round. This scheme can be seen in Fig. 5.

For S-Boxless version of implementation, the only difference is the additional cycle for reading S-Box values from memory, which can be seen in Fig. 6. Therefore, the total cycle count of state processing in each round is 28 cycles.

B. Key Schedule

At the end of every round, key scheduling of PRESENT is performed. Again, the first address (for first round, K0) is read to appear in the following cycle. Then, it is shifted by 3 bits and stored in the corresponding address in order to perform key scheduling. At the same time, least significant 3 bits of the read key data is stored in a 3-bit key temporary register for the shifting part of scheduling. The value kept in
key temporary register is also written into the corresponding address with the read and 3-bit shifted key data at the same cycle. This process is performed for all 8 addresses, which results in 10 cycles as seen in Fig. 7.

For S-Boxless version, again the S-Box value read part is added to this flow, which results in an increase of 2 cycles as shown in Fig. 8.

C. Final Round

At the last round, Round-32, only XOR operation is performed and the result is written-back to the original input addresses. To perform this, an XOR and write cycle comes after every read data and key read. This results in 8 cycles for both approaches, as there is no S-Box operation in this phase. This scheme is shown in Fig. 9.

D. Overall Design

According to the data flows explained above, the hardware of the design is defined. As can be seen from Fig. 10, the regular with S-Box implementation has rather simple logic; however, it uses 4 S-Boxes that increase the slice count. In Fig. 11, the block diagram of S-Boxless design is shown. It is easy to see that the control logic of this implementation is more complex, compared to the first approach. Therefore, even though it has no on-slice S-Boxes, it has even higher slice count than the first approach.

V. RESULTS AND DISCUSSION

Table I shows the performance comparison of both designs against prior work. As shown in the table, our designs achieve the lowest slice counts with the addition of a block RAM. On the cycle count, our designs are slower, but still offer respectable throughput for lightweight applications. Our figure of merit (throughput per slice) can reach one third of the previous most compact work.

VI. CONCLUSION AND FUTURE WORK

As the cost of FPGAs get lower together with their power consumption, they become more and more attractive

<table>
<thead>
<tr>
<th>PH</th>
<th>CYC</th>
<th>PORT-A OPERATION</th>
<th>PORT-A OUTPUT</th>
<th>TMP REGISTER</th>
<th>PORT-B OPERATION</th>
<th>PORT-B OUTPUT</th>
<th>PORT-B INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>READ S0</td>
<td>READ K0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>READ S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>READ S5</td>
<td></td>
<td>b12 b11 b10 b9</td>
<td>WRITE S4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>READ S6</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>READ S7</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. PRESENT cycles for the state processing phases of each round (on-slice S-box version).

<table>
<thead>
<tr>
<th>PH</th>
<th>CYC</th>
<th>PORT-A OPERATION</th>
<th>PORT-A OUTPUT</th>
<th>TMP REGISTER</th>
<th>PORT-B OPERATION</th>
<th>PORT-B OUTPUT</th>
<th>PORT-B INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>READ S0</td>
<td>READ K0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>READ S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>READ S5</td>
<td></td>
<td>b12 b11 b10 b9</td>
<td>WRITE S4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>READ S6</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>READ S7</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. PRESENT cycles for the first state processing phase of each round (on-RAM S-box version).
alternatives for lightweight applications such as sensor nodes and RFIDs. Most of these applications require security and suffer from limited resources on low-cost FPGAs. Therefore, it is crucial to implement lightweight cryptography on such devices in a way to ensure maximum possible amount of resources left for other circuitry that share the device. Until now, research has been done on register based implementations on FPGAs, but none of these studies considered utilization of the already existing block RAMs in order to minimize register consumption.

In this study, we present two different RAM based implementations of the lightweight cipher algorithm PRESENT on FPGAs. With their lowest reported slice counts, both implementations prove to be suitable for lightweight applications such as sensor nodes and RFIDs. While the first implementation uses slices for S-boxes, the second one integrates the S-boxes into the block RAM as well. This way, we provide two alternative platforms for future side-channel attack (SCA) resistant implementations.

The on-slice and on-RAM S-box based designs are perfect candidates for the application of the shared S-box technique in [12] and the block memory content scrambling technique [13] for SCA resistance, respectively. As the next step of this study, we shall apply these SCA measures on our designs,

![Figure 7](image1.png)

**Figure 7.** PRESENT cycles for the key expansion phase of each round (on-slice S-box version).

![Figure 8](image2.png)

**Figure 8.** PRESENT cycles for the key expansion phase of each round (on-RAM S-box version).

![Figure 9](image3.png)

**Figure 9.** PRESENT cycles for the final round (the same in both on-slice and on-RAM S-box versions).
Table I

<table>
<thead>
<tr>
<th>Design</th>
<th>Key size</th>
<th># of slices &amp; BRAMs</th>
<th>Cycle count</th>
<th>Tput@100KHz (Kbps)</th>
<th>Tput/slice@max.freq. (Mbps/slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our work</td>
<td>128</td>
<td>83 / 1</td>
<td>128</td>
<td>129.77</td>
<td>24.96</td>
</tr>
<tr>
<td>(on-slice S-boxes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our work</td>
<td>128</td>
<td>85 / 2</td>
<td>128</td>
<td>135.15</td>
<td>24.96</td>
</tr>
<tr>
<td>(on-RAM S-boxes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and observe their effects on the area as well as resistance to SCA attacks.

REFERENCES


