UNIVERSITY OF BIRMINGHAM

Secure Software and Hardware Systems

Assignment Two: PRESENT implementations

Student: Romualdas Paskevicius
(assignment completed individually)

1. PRESENT implementation and optimisation

1. 1. Reference Implementation

Reference implementation of PRESENT is available in ***present_ref/crypto.c***. The implementation has been tested against the provided test vectors, all of the tests passed. The test output may be seen in *Figure 1: Reference implementation results*.

- The average number of cycles per PRESENT execution: (82921 + 73592 + 72986 + 72987 + 72886) / 5 = 75074 cycles/execution.
- The encryption throughput in cycles per bit: 75074 cycles for 64 bit, therefore the throughput is 75074 / 64 = 1173.03125 cycles/bit.

```
C:\Users\Romas\Downloads\sechard\assignment2\present_ref>python test_against_testvectors.py COM3
[+] Using com port COM3
== Testvector 0
[+] Cycle count = 82921 = 0.000663368 s
[OK] Result correct: 45 84 22 7B 38 C1 79 55

== Testvector 1
[+] Cycle count = 73592 = 0.000588736 s
[OK] Result correct: 49 50 94 F5 C0 46 2C E7

== Testvector 2
[+] Cycle count = 72986 = 0.000583888 s
[OK] Result correct: 7B 41 68 2F C7 FF 12 A1

== Testvector 3
[+] Cycle count = 72987 = 0.000583896 s
[OK] Result correct: D2 10 32 21 D3 DC 33 33

== Testvector 4
[+] Cycle count = 72886 = 0.000583088 s
[OK] Result correct: D0 44 6A 0A C9 13 35 D4
```

Figure 1: Reference implementation results

It is worth noting that the first PRESENT execution immediately after the Pi Pico is programmed takes longer than subsequent executions. Therefore, more accurate performance rates have been calculated using repeated executions. As such, the reference implementation takes 74233 cycles/execution on average, and offers a throughput of 1159.890625 cycles/bit.

Special techniques/optimisations were not used in the reference implementation. This report aimed to contrast the naive reference implementation with a highly optimised bit-sliced implementation, therefore optimisations have been left to the bit-sliced implementation. However, the reference implementation could benefit from a number of optimisations, namely 8-bit SBoxes, and SPBoxes.

By default, PRESENT uses 4-bit SBoxes, and although these take up very little space, it is possible to exchange them for larger 8-bit SBoxes. This is known as the time/memory trade-off.

The larger 8-bit SBoxes would take up more memory, but result in faster execution times, as it would no longer be necessary to perform the bit shifting to align the state into 4-bit SBoxes.

In addition, PRESENT contains a permutation layer, which shuffles individual bits within the state. This proves to be slow, as the processor does not have the capabilities to address memory at the bit level, therefore a lot of bit shifting is needed to extract individual bits. As such, SPBoxes store already permuted bits, which avoids most of the bit shifting and greatly speeds up the execution times.

1. 2. Bitslicing Implementation

Bitslicing Implementation of PRESENT is available in ***present_bs/crypto.c***. The implementation has been tested against the provided test vectors, all of the tests passed. The test output from the final/optimised PRESENT implementation may be seen in *Figure 2: Bitslicing implementation results.*

```
C:\Users\Romas\Downloads\sechard\assignment2\present_bs>python test_against_testvectors.py COM3
== Key 0 = 00000000000000000000
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25]
[26] [27] [28] [29] [30] [31]
[+] Cycle count = 190942 = 0.001527536 s
[+] Cycle count per block = 5966.9375 = 4.77355e-05 s
[OK] Result correct

== Key 1 = ffffffffffffffffffff
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25]
[26] [27] [28] [29] [30] [31]
[+] Cycle count = 190637 = 0.001525096 s
[+] Cycle count per block = 5957.40625 = 4.765925e-05 s
[OK] Result correct

== Key 2 = 3cf400d828f1087a6026
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25]
[26] [27] [28] [29] [30] [31]
[+] Cycle count = 189884 = 0.001519072 s
[+] Cycle count per block = 5933.875 = 4.7471e-05 s
[OK] Result correct
```

Figure 2: Bitslicing implementation results.

In order to obtain a highly performant (in terms of speed) implementation of bitsliced PRESENT, a number of optimisations have been made. Initially, the bitsliced implementation calculated SBoxes by extracting individual bits and later recombining them, which could be seen in *Figure 3: Initial bitslicing implementation.*

```
46  static bs_reg_t sbox(uint8_t sbox_bitnum, bs_reg_t in0, bs_reg_t in1, bs_reg_t in2, bs_reg_t in3)
47  {
48      bs_reg_t out = 0;
49
50      for (uint8_t bit = 0; bit < BITSLICE_WIDTH; bit++)
51      {
52          uint8_t x0 = (in0 >> bit) & 1;
53          uint8_t x1 = (in1 >> bit) & 1;
54          uint8_t x2 = (in2 >> bit) & 1;
55          uint8_t x3 = (in3 >> bit) & 1;
56
57          uint8_t y = 0;
58
59          if (sbox_bitnum == 0)
60          {
61              y = x0 ^ (x1 & x2) ^ x2 ^ x3;
62          }
76          out |= (y << bit);
77
78      }
79
80      return out;
81  }
```

Figure 3: Initial bitslicing implementation.

However, extracting individual bits and later recombining them resulted in very poor performance, even slower than the naive PRESENT implementation. Such an approach took 84152 cycles/execution of one 64-bit instance, and provided a throughput of 1314.875 cycles/bit. Therefore, it proved essential to optimise the SBox computation.

It is worth noting that all of the operations used within the SBox are bitwise operations, therefore it proved that extracting individual bits is not necessary. As such, the SBox computation has been modified to directly perform the bitwise operations without extracting individual bits as seen in *Figure 4: Bitslicing implementation: faster SBox computation.* Such an approach provided a major speed improvement: 6073 cycles/execution of one 64-bit instance, and throughput of 94.890625 cycles/bit.

```
46  static bs_reg_t sbox0(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
47  {
48      return x0 ^ (x1 & x2) ^ x2 ^ x3;
49  }
50
51  static bs_reg_t sbox1(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
52  {
53      return (x0 & x2 & x1) ^ (x0 & x3 & x1) ^ (x3 & x1) ^ x1 ^ (x0 & x2 & x3) ^ (x2 & x3) ^ x3;
54  }
55
56  static bs_reg_t sbox2(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
57  {
58      return (x0 & x1) ^ (x0 & x3 & x1) ^ (x3 & x1) ^ x2 ^ (x0 & x3) ^ (x0 & x2 & x3) ^ x3 ^ 0xFFFFFFFF;
59  }
60
61  static bs_reg_t sbox3(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
62  {
63      return (x1 & x2 & x0) ^ (x1 & x3 & x0) ^ (x2 & x3 & x0) ^ x0 ^ x1 ^ (x1 & x2) ^ x3 ^ 0xFFFFFFFF;
64  }
```

Figure 4: Bitslicing implementation: faster SBox computation.

In addition, the improved SBox could be optimised even further. The bitwise operations had a lot of redundancy, as the same calculations were often performed twice. Therefore, it is possible to rearrange the computations to improve the performance of bitsliced PRESENT. Two approaches have been used to tackle the redundancy issue: rearranging and storing intermediate computations.

Firstly, it is possible to apply rearranging to computations of y1 and y3:

$$y1 = x0 \cdot x2 \cdot x1 + x0 \cdot x3 \cdot x1 + x3 \cdot x1 + x1 + x0 \cdot x2 \cdot x3 + x2 \cdot x3 + x3$$
$$y1 = x0 \cdot x1 \cdot (x2 + x3) + x3 \cdot x1 + x1 + x0 \cdot x2 \cdot x3 + x2 \cdot x3 + x3$$

$$y3 = x1 \cdot x2 \cdot x0 + x1 \cdot x3 \cdot x0 + x2 \cdot x3 \cdot x0 + x0 + x1 + x1 \cdot x2 + x3 + 1$$
$$y3 = x1 \cdot x2 \cdot x0 + x3 \cdot x0 \cdot (x1 + x2) + x0 + x1 + x1 \cdot x2 + x3 + 1$$

Secondly, it is possible to store intermediate computations in y1, y2 and y3:

Assume $c1 = x2 \cdot x3$, then:

$$y1 = x0 \cdot x1 \cdot (x2 + x3) + x3 \cdot x1 + x1 + x0 \cdot x2 \cdot x3 + x2 \cdot x3 + x3$$
$$y1 = x0 \cdot x1 \cdot (x2 + x3) + x3 \cdot x1 + x1 + x0 \cdot c1 + c1 + x3$$

Assume $c2 = x0 \cdot x3$, then:

$$y2 = x0 \cdot x1 + x0 \cdot x3 \cdot x1 + x3 \cdot x1 + x2 + x0 \cdot x3 + x0 \cdot x2 \cdot x3 + x3 + 1$$
$$y2 = x0 \cdot x1 + c2 \cdot x1 + x3 \cdot x1 + x2 + c2 + c2 \cdot x2 + x3 + 1$$

Assume $c3 = x1 \cdot x2$, then:

$$y3 = x1 \cdot x2 \cdot x0 + x3 \cdot x0 \cdot (x1 + x2) + x0 + x1 + x1 \cdot x2 + x3 + 1$$
$$y3 = c3 \cdot x0 + x3 \cdot x0 \cdot (x1 + x2) + x0 + x1 + c3 + x3 + 1$$

With the improved SBox equations implemented, the encryption performed even faster: 5952.7375 cycles/execution of one 64-bit instance, and throughput of 93.01 cycles/bit. The final SBox computation code may be seen in *Figure 5: Bitslicing implementation: final SBox computation.*

```
#define BS_INV 0xFFFFFFFF

static bs_reg_t sbox0(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
{
    return x0 ^ (x1 & x2) ^ x2 ^ x3;
}

static bs_reg_t sbox1(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
{
    bs_reg_t c = x2 & x3;
    return ((x0 & x1) & (x2 ^ x3)) ^ (x3 & x1) ^ x1 ^ (x0 & c) ^ c ^ x3;
}

static bs_reg_t sbox2(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
{
    bs_reg_t c = x0 & x3;
    return (x0 & x1) ^ (c & x1) ^ (x3 & x1) ^ x2 ^ c ^ (c & x2) ^ x3 ^ BS_INV;
}

static bs_reg_t sbox3(bs_reg_t x0, bs_reg_t x1, bs_reg_t x2, bs_reg_t x3)
{
    bs_reg_t c = x1 & x2;
    return (c & x0) ^ ((x3 & x0) & (x1 ^ x2)) ^ x0 ^ x1 ^ c ^ x3 ^ BS_INV;
}
```

Figure 5: Bitslicing implementation: final SBox computation.

Detailed performance comparison may be seen in *Figure 6: Cycles Per Execution vs Implementation* and *Figure 7: Throughput (Cycles/Bit) vs Implementation.*

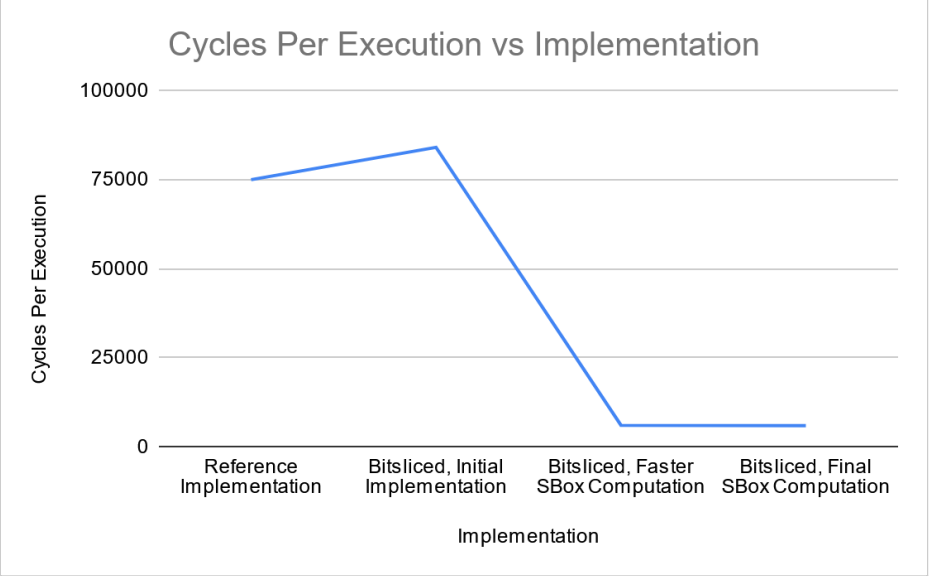

Figure 6: Cycles Per Execution vs Implementation

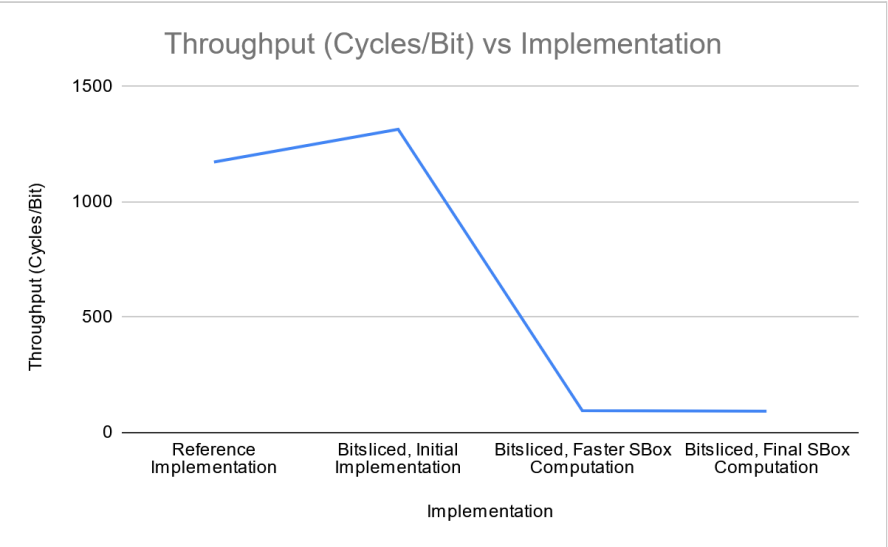

Figure 7: Throughput (Cycles/Bit) vs Implementation

With rearranging and intermediate computations implemented, the final bitsliced implementation performed very well, offering more than 10 times speed improvements to the naive implementation of PRESENT.

- The average number of cycles per PRESENT execution: (190942 + 190637 + 189884) / 3 = 190487.6 cycles/execution. However, each execution computes 32 instances at once, therefore the average number of cycles per execution of one instance is 190487.6 / 32 = 5952.7375 cycles/execution (more than 10 times less cycles than the naive implementation of PRESENT).
- The encryption throughput in cycles per bit: 5952.7375 cycles for 64 bit, therefore the throughput is 5952.7375 / 64 = 93.01 cycles/bit (more than 10 times less cycles than the naive implementation of PRESENT).

Statement of good academic conduct

By submitting this assignment, I understand that I am agreeing to the following statement of good academic conduct:

- I confirm that this assignment is **my own work** and I have not worked with others in preparing this assignment.
- I confirm this assignment was written by me and is in my own words, except for any materials from published or other sources which are clearly indicated and acknowledged as such by appropriate referencing.
- I confirm that this work is not copied from any other person's work (published or unpublished), web site, book or other source, and has not previously been submitted for assessment either at the University of Birmingham or elsewhere.
- I confirm that I have not asked, or paid, others to prepare any part of this work for me.
- I confirm that I have read and understood the University regulations on plagiarism (https://intranet.birmingham.ac.uk/as/registry/policy/conduct/plagiarism/index.aspx).