

Professor: David Oswald

Assignment 2 Report

Secure Software and Hardware

Names: James Gwynn, Shinya Sano, and
Mihai Stavila

Group Name: Project Groups Ex2 2

Word Count: 986

Contents

Reference Implementation	2
PRESENT executions	2
Number of cycles	2
Throughput in cycles per bit	2
Bitslicing Implementation	2
Optimizations and Effect on Performance	2
Results comparison overview	2
Algebraic Normal Form (ANF)	2
Unrolled loops	2
Minimised S-Box Boolean expressions	3
Optimized PRESENT executions	3
Number of cycles	3
Throughput in cycles per bit	3
Bibliography	4
References	5
Appendix A – Reference Implementation figures (PRESENT)	6
Appendix B – Bitslicing Implementation figures (PRESENT)	8

Reference Implementation

PRESENT executions

Number of cycles

As demonstrated in Figures 1-2 in appendix a, the number of clock cycles for each PRESENT execution varies. However, the common average for the overall cycle count mainly computes to around the range of 107749-107799 cycles per test. The general assumption is that it generates 68000 cycles per 8-byte plaintext in this experiment which represents the unoptimized version of cryptography in this experiment.

Throughput in cycles per bit

The throughput was calculated to get a full evaluation of the overall performance of the PRESENT code. For an isolated example, one test vector computed the total number of cycles to be 107851 as shown in Figure 3. The throughput is found by dividing the cycles by the sum of bits from each 8-byte plain-text block which is 64 bits. The result of this would be around 1,685 cycles per bit. This information gives us a full idea of how much computation power is being used. As proven in Figure 4, PRESENT has a relatively high throughput compared to other cryptographic algorithms like AES (Pei et al, 2018). Fast throughput is generally desirable as it allows for fast encryption rates in the total amount of cycles.

Bitslicing Implementation

Optimizations and Effect on Performance

Results comparison overview

By using some optimization techniques, the number of cycles required to compute the PRESENT algorithm was able to be reduced, saving time and power. The code for the bitsliced implementation of PRESENT went through several variations before deciding on a finalised solution that provided the most efficient implementation.

Algebraic Normal Form (ANF)

As presented in appendix b figures 5-7, the first variation of our bitsliced implementation code already has reduced the number of cycles compared to the reference execution of PRESENT. This is partly due to our use of Boolean expressions in the S-Box to generate the outputs. ^ represents XOR and & represents AND, these Boolean functions are represented as ciphertext bits whilst the variables are the plaintext bits and key bits. The purpose is to find the XOR-sum regarding the ciphertext, where each bit is regarded as an XOR sum of bits from the plaintext and the key used for encryption. This impacts performance as this decreases the amount of time required to search for the required keys and efficiently schedule how they are used for the output of ciphertext. Figure 8 represents the S-Box ANF for the first variation of the bitsliced PRESENT code (Peng et al, 2012).

Unrolled loops

Optimising the enslice and unslice functions was a main contributor to further reducing computation time and creating a more effective performance in our implementation. This was done by unrolling the loops. Figures 9-10 indicate that each 8-byte plaintext has its individual bits represented in bitsliced form in the enslice function from the revised code,

compared to a nested for loop which doesn't present the iteration through each input bit. Figures 11-12 also do the same process but convert bitsliced bit states into normal plaintext.

This is a useful method as loop unrolling reduces power consumption in dynamic programming by decreasing overhead. The overhead buffer when either encrypting or decrypting plaintext increases the computation time which maximises the number of clock cycles required (Abolade et al, 2021). Unrolling loops allows more clear instructions regarding how the compiler can schedule across the loops which gives less of a reason to allow overhead when transferring data (Texas Instruments, 2020). Figures 13-15 also show how there are much fewer overall cycles per 8-byte plaintext block compared to both the reference implementation and the first variation of the bitslicing implementation. Partly due to this method being utilised.

Minimised S-Box Boolean expressions

Another method to create a more effective performance with fewer clock cycles was to minimise the number of Boolean expressions for the S-Box. The cycles are reduced as this makes the encryption process of plaintext to ciphertext more easily manageable and faster. Figure 16 shows how the ANF has been updated in the final version where much less XORs and ANDs are used. This is responsible for using less computation time in the trapdoor round function, especially as it is non-linear and allows each 8-byte to input 4 bits of plaintext to generate 4 bits of ciphertext in parallel with one another (Rijmen and Preneel, 1997). Overall, this is a good contribution that has enabled around 400 fewer cycles compared to the first variation without this method.

Optimized PRESENT executions

Number of cycles

On average, there are more overall cycles in both the first and final versions of our bitslicing implementation compared to the reference application. This is due to 32 instances of converting the plaintext into bitsliced form in the `enslice` function but also converting it back in the `unslice` function. This is iterated through each bit in the four 8-byte plaintexts provided.

The average number of overall cycles per execution in the final optimised version is predicted to be around either 164020 or 165648 as they are the most frequent numbers. The execution tests also show that the average of cycles per 8-byte plaintext block is either 5125.625 or 5176 in a typical test.

Throughput in cycles per bit

We divided the number of cycles by the number of bits in an 8-byte plaintext, which in this case is 64 bits. Therefore, the throughput should be at least approximately 2,562 per bit for the overall throughput. At most, the overall cycle throughput is predictably 2,933 cycles per bit as the highest overall cycle count during testing was 187725. For the 8-byte plaintext block, the throughput is at least 80 cycles per bit and at most 92 cycles per bit due to the highest number of cycles for the blocks in testing being 5866.40625. This formula is utilised in other implementations using known block ciphers like SPEEDY (GitHub, 2023) where it achieved similar results in cycles per byte in Figure 17 (Kim et al, 2022).

Bibliography

Abolade, O., Okandeji, A., Oke, A., Osifeko, M. and Oyedeji, A. (2021) 'Overhead effects of data encryption on TCP throughput across IPSEC secured network', *Scientific African*, 13, pp. 1-6.

Kim, H., Eum, S., Sim, M. and Seo, H. (2022) 'Efficient Implementation of SPEEDY Block Cipher on Cortex-M3 and RISC-V Microcontrollers', *Mathematics*, 10(22), pp. 1-12.

Pei, C., Xiao, Y., Liang, W. and Han, X. (2018) 'Trade-off of security and performance of lightweight block ciphers in Industrial Wireless Sensor Networks', *Journal on Wireless Communication and Networking*, 117, pp. 1-18.

Peng, C., Zhu, C., Zhu, Y. and Kang, F. (2012) 'Symbolic computation in block cipher with application to PRESENT', *Cryptology ePrint Archive*, pp. 1-11. Available at: <https://ia.cr/2012/587>

Rijment, V. and Preneel, B. (1997) 'A Family of Trapdoor Ciphers', *Fast Software Encryption FSE'97*, Haifa, Israel, 20-22 January 1997. Available at: <https://doi.org/10.1007/BFb0052342> (Accessed: 27 April 2023).

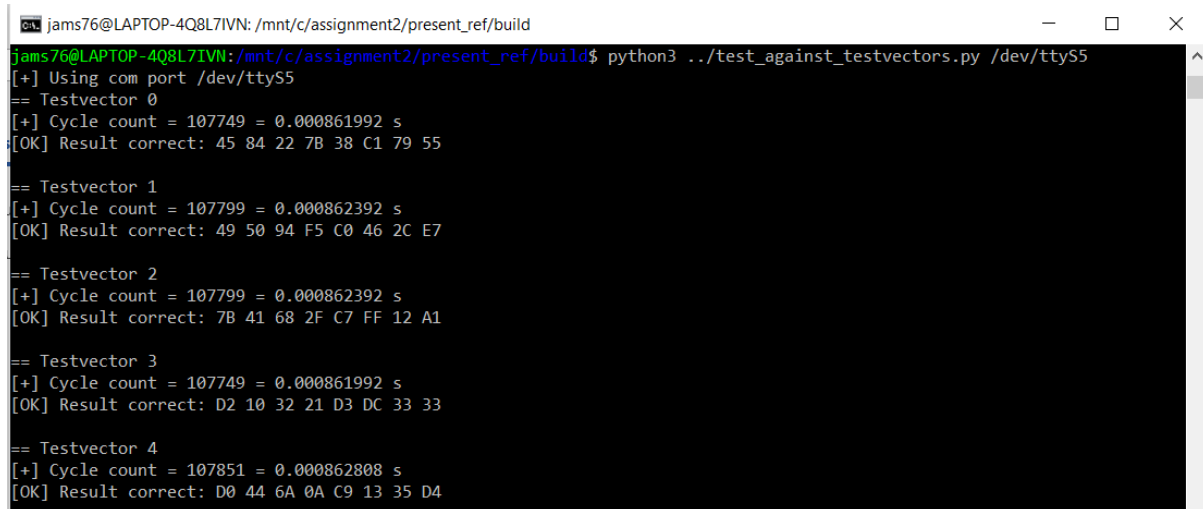
References

GitHub (2023) *SPEEDY*. Available at: <https://github.com/Chair-for-Security-Engineering/SPEEDY> (Accessed: 28 April 2023).

Texas Instruments (2020) 4.7. *Loop unrolling*. Available at: https://software-dl.ti.com/C2000/docs/optimization_guide/phase3/loop_unrolling.html#:~:text=Improved%20floating%2Dpoint%20performance%20%2D%20loop,opportunity%20to%20generate%20parallel%20instructions. (Accessed: 27 April 2023).

Appendix A – Reference Implementation figures (PRESENT)

Figure 1: Reference execution test (1)



```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_ref/build
jams76@LAPTOP-4Q8L7IVN:/mnt/c/assignment2/present_ref/build$ python3 ../test_against_testvectors.py /dev/ttyS5
[+] Using com port /dev/ttyS5
== Testvector 0
[+] Cycle count = 107749 = 0.000861992 s
[OK] Result correct: 45 84 22 7B 38 C1 79 55

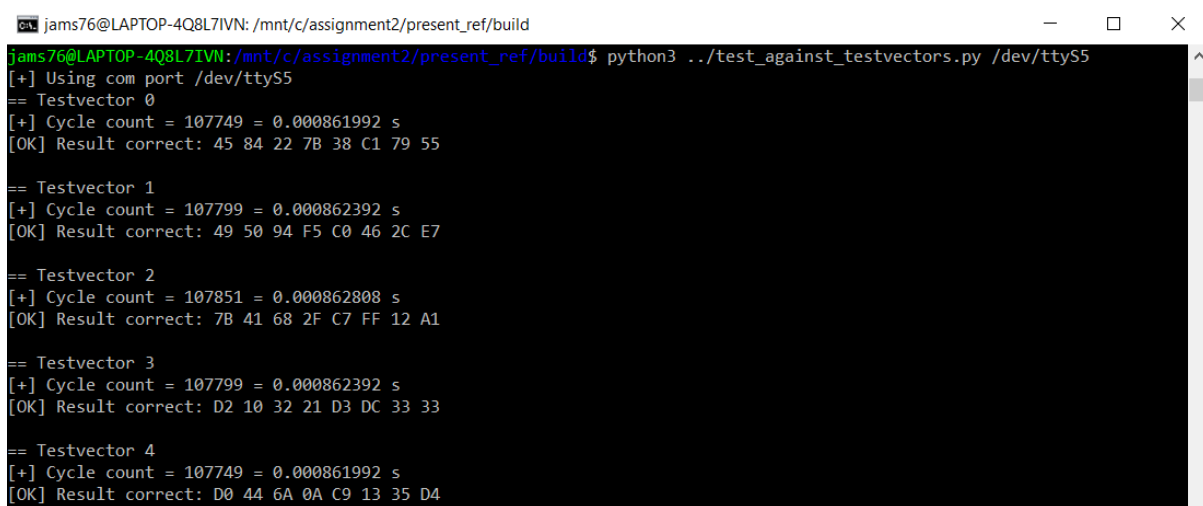
== Testvector 1
[+] Cycle count = 107799 = 0.000862392 s
[OK] Result correct: 49 50 94 F5 C0 46 2C E7

== Testvector 2
[+] Cycle count = 107799 = 0.000862392 s
[OK] Result correct: 7B 41 68 2F C7 FF 12 A1

== Testvector 3
[+] Cycle count = 107749 = 0.000861992 s
[OK] Result correct: D2 10 32 21 D3 DC 33 33

== Testvector 4
[+] Cycle count = 107851 = 0.000862808 s
[OK] Result correct: D0 44 6A 0A C9 13 35 D4
```

Figure 2: Reference execution test (2)



```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_ref/build
jams76@LAPTOP-4Q8L7IVN:/mnt/c/assignment2/present_ref/build$ python3 ../test_against_testvectors.py /dev/ttyS5
[+] Using com port /dev/ttyS5
== Testvector 0
[+] Cycle count = 107749 = 0.000861992 s
[OK] Result correct: 45 84 22 7B 38 C1 79 55

== Testvector 1
[+] Cycle count = 107799 = 0.000862392 s
[OK] Result correct: 49 50 94 F5 C0 46 2C E7

== Testvector 2
[+] Cycle count = 107851 = 0.000862808 s
[OK] Result correct: 7B 41 68 2F C7 FF 12 A1

== Testvector 3
[+] Cycle count = 107799 = 0.000862392 s
[OK] Result correct: D2 10 32 21 D3 DC 33 33

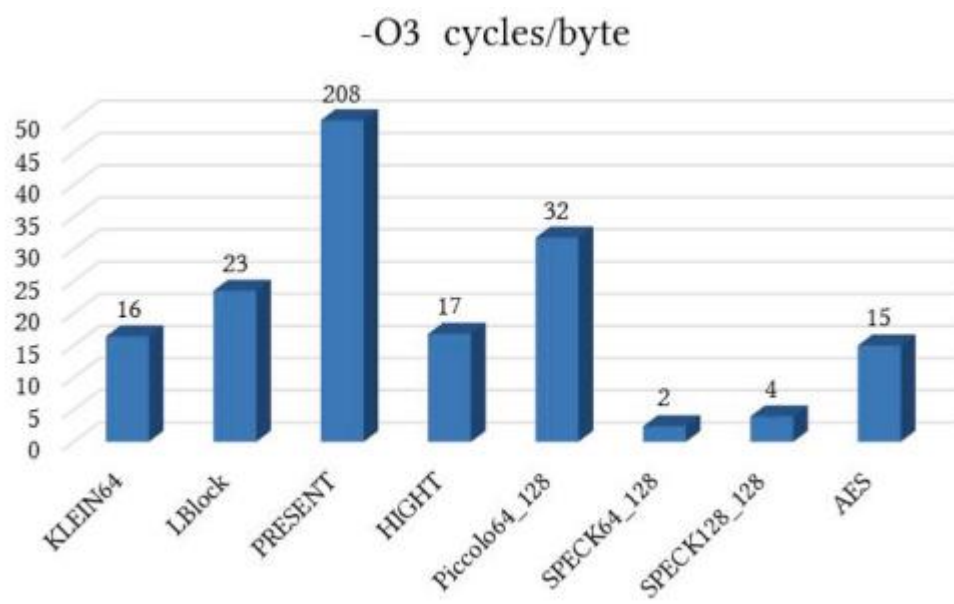
== Testvector 4
[+] Cycle count = 107749 = 0.000861992 s
[OK] Result correct: D0 44 6A 0A C9 13 35 D4
```

Figure 3: Test vector with 107851 cycle count



```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_ref/build
== Testvector 2
[+] Cycle count = 107851 = 0.000862808 s
[OK] Result correct: 7B 41 68 2F C7 FF 12 A1
```

Figure 4: Cryptography algorithm comparison (cycles/bytes)



Appendix B – Bitslicing Implementation figures (PRESENT)

Figure 5: Unoptimized bitslicing execution test (1)

```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build
jams76@LAPTOP-4Q8L7IVN:/mnt/c/assignment2/present_bs/build$ python3 ../test_against_testvectors.py /dev/ttyS5
== 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 = 18446744073692959051 = 147573952589.54367 s
[+] Cycle count per block = 5.7646075230290496e+17 = 4611686018.42324 s
[OK] Result correct

== Key 1 = ffffffffffffffffffffff
[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 = 177554 = 0.001420432 s
[+] Cycle count per block = 5548.5625 = 4.43885e-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 = 175773 = 0.001406184 s
[+] Cycle count per block = 5492.90625 = 4.394325e-05 s
[OK] Result correct
```

Figure 6: Unoptimized bitslicing execution test (2)

```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build
jams76@LAPTOP-4Q8L7IVN:/mnt/c/assignment2/present_bs/build$ python3 ../test_against_testvectors.py /dev/ttyS5
== 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 = 175787 = 0.001406296 s
[+] Cycle count per block = 5493.34375 = 4.394675e-05 s
[OK] Result correct

== Key 1 = ffffffffffffffffffffff
[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 = 177300 = 0.0014184 s
[+] Cycle count per block = 5540.625 = 4.4325e-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 = 175672 = 0.001405376 s
[+] Cycle count per block = 5489.75 = 4.3918e-05 s
[OK] Result correct
```

Figure 7: Unoptimized bitslicing execution test (3)

```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build
jams76@LAPTOP-4Q8L7IVN:/mnt/c/assignment2/present_bs/build$ python3 ../test_against_testvectors.py /dev/ttyS5
== 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 = 175029 = 0.001400232 s
[+] Cycle count per block = 5469.65625 = 4.375725e-05 s
[OK] Result correct

== Key 1 = ffffffffffffffffffffff
[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 = 177300 = 0.0014184 s
[+] Cycle count per block = 5540.625 = 4.4325e-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 = 175672 = 0.001405376 s
[+] Cycle count per block = 5489.75 = 4.3918e-05 s
[OK] Result correct
```

Figure 8: ANF of unoptimized bitslicing execution

```
//original version
//b0 = a0 ^ (a1 & a2) ^ a2 ^ a3;
//b1 = (a0 & a2 & a1) ^ (a0 & a3 & a1) ^ (a3 & a1) ^ a1 ^ (a0 & a2 & a3) ^ (a2 & a3) ^ a3;
//b2 = ~((a0 & a1) ^ (a0 & a3 & a1) ^ (a3 & a1) ^ a2 ^ (a0 & a3) ^ (a0 & a2 & a3) ^ a3);
//b3 = ~((a1 & a2 & a0) ^ (a1 & a3 & a0) ^ (a2 & a3 & a0) ^ a0 ^ a1 ^ (a1 & a2) ^ a3);
```

Figure 9: Rolled loop- enslice function (First variation of bitsliced code)

```
static void enslice(const uint8_t pt[CRYPTO_IN_SIZE * BITSlice_WIDTH], bs_reg_t state_bs[CRYPTO_IN_SIZE_BIT])
{
    // INSERT YOUR CODE HERE AND DELETE THIS COMMENT
    uint8_t idx;

    for (idx = 0; idx < CRYPTO_IN_SIZE_BIT; idx++) {
        for (uint8_t slice = 0; slice < BITSlice_WIDTH; slice++) {
            bs_reg_t temp = (pt[slice * CRYPTO_IN_SIZE + idx / 8] >> (idx % 8)) & 0x1;
            state_bs[idx] |= temp << slice;
        }
    }
}
```

Figure 10: Unrolled loop- enslice function (Final variation of bitsliced code)

```
static void enslice(const uint8_t pt[CRYPTO_IN_SIZE * BITSlice_WIDTH], bs_reg_t state_bs[CRYPTO_IN_SIZE_BIT])
{
    // convert plaintext in normal byte form into bit-sliced form using bitwise operations
    // iterate through each input buffer bit, extracting bits from each pt array bytes and storing it
    // in the corresponding position in the bit-sliced register array
    for (uint8_t i = 0; i < CRYPTO_IN_SIZE_BIT; i++) {
        // Unrolled loop to potentially improve performance by reducing overhead
        // each line covers sets of 8 bits from the input normal form pt array
        // Convert normal buffer into bitsliced form via bitwise operations
        // Use bitwise OR between bit-sliced register array's value and the new bit
        // extracted from every byte inside the normal form input array;
        // bit pos is decided by i modulo 8, and the byte from which it is extracted is decided by i divided by 8
        state_bs[i] |= ((pt[0 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 0;
        state_bs[i] |= ((pt[1 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 1;
        state_bs[i] |= ((pt[2 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 2;
        state_bs[i] |= ((pt[3 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 3;
        state_bs[i] |= ((pt[4 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 4;
        state_bs[i] |= ((pt[5 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 5;
        state_bs[i] |= ((pt[6 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 6;
        state_bs[i] |= ((pt[7 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 7;

        state_bs[i] |= ((pt[8 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 8;
        state_bs[i] |= ((pt[9 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 9;
        state_bs[i] |= ((pt[10 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 10;
        state_bs[i] |= ((pt[11 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 11;
        state_bs[i] |= ((pt[12 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 12;
        state_bs[i] |= ((pt[13 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 13;
        state_bs[i] |= ((pt[14 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 14;
        state_bs[i] |= ((pt[15 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 15;

        state_bs[i] |= ((pt[16 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 16;
        state_bs[i] |= ((pt[17 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 17;
        state_bs[i] |= ((pt[18 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 18;
        state_bs[i] |= ((pt[19 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 19;
        state_bs[i] |= ((pt[20 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 20;
        state_bs[i] |= ((pt[21 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 21;
        state_bs[i] |= ((pt[22 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 22;
        state_bs[i] |= ((pt[23 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 23;

        state_bs[i] |= ((pt[24 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 24;
        state_bs[i] |= ((pt[25 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 25;
        state_bs[i] |= ((pt[26 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 26;
        state_bs[i] |= ((pt[27 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 27;
        state_bs[i] |= ((pt[28 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 28;
        state_bs[i] |= ((pt[29 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 29;
        state_bs[i] |= ((pt[30 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 30;
        state_bs[i] |= ((pt[31 * CRYPTO_IN_SIZE + i / 8] >> (i % 8)) & 1) << 31;
    }
}
```

Figure 11: Rolled loop- unslice function (First variation of bitsliced code)

```
static void unslice(const bs_reg_t state_bs[CRYPTO_IN_SIZE_BIT], uint8_t pt[CRYPTO_IN_SIZE * BITSlice_WIDTH])
{
    // INSERT YOUR CODE HERE AND DELETE THIS COMMENT
    uint8_t idx;

    for (idx = 0; idx < CRYPTO_IN_SIZE_BIT; idx++) {
        for (uint8_t slice = 0; slice < BITSlice_WIDTH; slice++) {
            uint8_t temp = (state_bs[idx] >> slice) & 0x1;
            pt[slice * CRYPTO_IN_SIZE + idx / 8] |= temp << (idx % 8);
        }
    }
}
```

Figure 12: Unrolled loop- unslice function (Final variation of bitsliced code)

Figure 14: Optimized bitslicing execution test (2)

```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build$ python3 ../test_against_testvectors.py /dev/ttyS5
== 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 = 164486 = 0.001315888 s
[+] Cycle count per block = 5140.1875 = 4.11215e-05 s
[OK] Result correct

== Key 1 = ffffffffffffffffffffff
[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 = 165648 = 0.001325184 s
[+] Cycle count per block = 5176.5 = 4.1412e-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 = 164020 = 0.00131216 s
[+] Cycle count per block = 5125.625 = 4.1005e-05 s
[OK] Result correct
```

Figure 15: Optimized bitslicing execution test (3)

```
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build
jams76@LAPTOP-4Q8L7IVN: /mnt/c/assignment2/present_bs/build$ python3 ../test_against_testvectors.py /dev/ttyS5
== Key 0 = 000000000000000000000000
[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 = 163376 = 0.001307008 s
[+] Cycle count per block = 5105.5 = 4.0844e-05 s
[OK] Result correct

== Key 1 = ffffffffffffffffffffff
[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 = 165648 = 0.001325184 s
[+] Cycle count per block = 5176.5 = 4.1412e-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 = 164020 = 0.00131216 s
[+] Cycle count per block = 5125.625 = 4.1005e-05 s
[OK] Result correct
```

Figure 16: ANF of optimized bitslicing execution

```
// S-Box layer to apply the s-box function to each block of 4 bits of the given state
// in order to ensure the encryption is not linear.
// 16 sbox lookups resulting since it is applied to each nibble of the state with 64 bits
for (uint8_t i = 0; i < 16; i++) {
    bs_reg_t a0, a1, a2, a3, b0, b1, b2, b3;

    a0 = state[i * 4];
    a1 = state[i * 4 + 1];
    a2 = state[i * 4 + 2];
    a3 = state[i * 4 + 3];

    // Minimized the S-Box expressions to improve computation speed thanks to fewer terms
    b0 = a0 ^ (~a1 & a2) ^ a3;
    b1 = (a0 & (a1 & a2)) ^ (~a0 & (a3 & (a1 ^ a2))) ^ (a1 ^ a3) ;
    b2 = (a0 & (a3 & (a1 ^ a2))) ^ (a0 & (a1 ^ a3)) ^ (~a1 & a3) ^ ~a2;
    b3 = (~a0 & ~(a1 & a2)) ^ (a0 & (a3 & (a1 ^ a2))) ^ (a1 ^ a3);

    // fixed 4 to 4-bit substitution (4 bit input & output)
    state[i * 4] = b0;
    state[i * 4 + 1] = b1;
    state[i * 4 + 2] = b2;
    state[i * 4 + 3] = b3;
}
```

Figure 17: Present vs Bitslicing (Clock Cycles per byte)

Intel 8th Core i7-8850H	Speed (cpb)
SPEEDY-7-192 encryption reference [9]	2983
6×32 reference [9]	1278
bitslice(our)	852