Lightweight Cryptography for the Cloud: Exploit the Power of Bitslice Implementation

Seiichi Matsuda[1] and Shiho Moriai[2]

[1] Sony Corporation
1-7-1 Konan, Minato-ku, Tokyo 108-0075, Japan
SeiichiA.Matsuda@jp.sony.com
[2] National Institute of Information and Communications Technology (NICT)
4-2-1 Nukui-Kitamachi, Koganei, Tokyo 184-8795, Japan
shiho.moriai@nict.go.jp

Abstract. This paper shows the great potential of lightweight cryptography in fast and timing-attack resistant software implementations in cloud computing by exploiting bitslice implementation. This is demonstrated by bitslice implementations of the PRESENT and Piccolo lightweight block ciphers. In particular, bitsliced PRESENT-80/128 achieves 4.73 cycles/byte and Piccolo-80 achieves 4.57 cycles/byte including data conversion on an Intel Xeon E3-1280 processor (Sandy Bridge microarchitecture). It is also expected that bitslice implementation offers resistance to side channel attacks such as cache timing attacks and cross-VM attacks in a multi-tenant cloud environment. Lightweight cryptography is not limited to constrained devices, and this work opens the way to its application in cloud computing.

Keywords: lightweight cryptography, software implementation, bitslice implementation, cloud, block cipher, PRESENT, Piccolo

1 Introduction

The cyber physical system has emerged as a promising direction for enriching interactions between physical and virtual worlds [14]. Many wireless sensor networks, for instance, monitor some aspect of the environment or human behaviors, and relay the data to the *cloud* for processes such as data mining, business intelligence and predictive analytics. Preservation of security and privacy in the sensed information in this system is essential.

Lightweight cryptography, which can be implemented on resource-constrained devices, is attracting attention for protecting private and sensitive information gathered on *sensors*. Recently many lightweight cryptographic primitives have been proposed, such as block ciphers, stream ciphers, hash functions, message-authentication codes [3, 19, 5, 1, 7, 8, 20]. Moreover, an international standard of lightweight cryptography (ISO/IEC 29192) has been developed in ISO/IEC JTC 1/SC 27.

Most lightweight cryptographic algorithms are designed to minimize the resource consumption of a *hardware implementation* such as area, power, and energy consumption, while some are software-oriented with design criteria such as low memory requirements, small code size, and limited instruction sets for low-end (e.g., 8-bit) platforms. As a result of design trade-off, some of lightweight cryptographic algorithms do not show good throughput in software implementation on mid-range to high-end microprocessors (e.g., Intel Core i7 processors) typically used for cloud computing.

This paper shows the great potential of lightweight cryptography in *fast* software implementations in cloud environments by exploiting bitslice implementation demonstrated through bitslice implementations of PRESENT and Piccolo. PRESENT and Piccolo are lightweight 64-bit block ciphers: the former was presented by Bogdanov et al. at CHES 2007 [3] and is specified in ISO/IEC 29192-2 [11], and the latter was presented by Shibutani et al. at CHES 2011 [20]. In particular, PRESENT-80/128 achieves 4.73 cycles/byte and Piccolo-80 achieves 4.57 cycles/byte on an Intel Xeon E3-1280 processor (Sandy Bridge). PRESENT-80/128 achieves 5.79 cycles/byte. Piccolo-80 achieves 5.69 cycles/byte on an Intel Core i7 870 (Nehalem), which are faster than the bitsliced AES's fastest record on the same microarchitecture (6.92 cycles/byte on an Intel Core i7 920) [12].

Only a few *software* performance data of lightweight cryptography for comparison exist in public literature. As for PRESENT, in [17] there are some software implementation results on 4-bit, 8-bit and 16-bit microcontrollers, but on a 32-bit processor, the encryption speed available is 16.2 cycles/byte on a Pentium III. In [9], it is written that optimized table-based implementations run 57 and 86 cycles/byte on a Core i7 Q720 for LED-64 and LED-128, respectively, and that they are faster than PRESENT. There is also previous work on bitsliced PRESENT by Grabher et al. [6], but their implementation results are not competitive.

It has been known that bitslice implementation is also resistant to cache timing attacks because it has no table lookups. In a multi-tenant cloud environment, cross-virtual machine (VM) attacks become new threats [18]. Bitslice implementation mitigates these risks.

The remainder of this paper is structured as follows. Section 2 shows a brief history of bitslice implementation, a use case of lightweight block ciphers in the cloud, and our target. Sections 3 and 4 respectively show bitslice implementations of lightweight block ciphers PRESENT and Piccolo, including optimizing techniques. Section 5 shows performance data and comparison with previous results, and Section 6 gives our conclusion.

2 Bitslice Implementation

Biham in 1997 introduced bitslicing as a technique for implementing cryptographic algorithms to improve the software performance of DES [2]. It was implemented on several processors and used for brute force key search of DES in the DES Challenges project in the late-1990s. The basic concept of bitslicing is

Fig. 1. Use case of decryption in bitslice implementation in the cloud

to simulate a hardware implementation in software. The entire algorithm is represented as a sequence of logical operations. On a processor with n-bit registers, a logical instruction corresponds to simultaneous execution of n hardware logical gates. In the bitslice implementation, S-boxes are computed using bit-logical instructions rather than table lookups. Since the execution time of these instructions is independent of the input and key values, the bitslice implementation is generally resistant to timing attacks.

Bitslice implementation techniques have progressed. The bitslice implementations of block ciphers presented by Biham were to encrypt/decrypt independent n blocks on a processor with n-bit registers. Matsui and Nakajima [15] demonstrated remarkable performance gain on Intel's Core 2 processor by fully utilizing its enhanced SIMD architecture. They showed a bitsliced AES running at the speed of 9.2 cycles/byte on a Core 2, which was faster than any previous standard table-based implementations. A hurdle in this implementation was that as many as n independent blocks needed to be processed simultaneously. Könighofer [13] presented an alternative implementation for 64-bit platforms that processes only four input blocks in parallel. Käsper and Schwabe [12] extended this approach and achieved a bitsliced AES in counter mode running at 7.59 cycles/byte on a Core 2.

A Use case of lightweight block ciphers in the cloud. In cyber physical systems, analyzing large data sets – so-called *big data* – will become a key basis of competition, underpinning new waves of productivity growth, innovation, and consumer surplus. Cloud computing will play an important role in analyzing big data, where scale-out software systems running on low-cost "commodity" platforms are expected. When sensor data need to be encrypted for privacy protection, encryption on a low-cost embedded hardware module using a lightweight block cipher will be the most cost competitive solution on the sensor side. Encrypted sensor data are collected from many sensors and decrypted on servers in the cloud when needed.

Bitslice implementation provides leverage in this use case. In most cases it can be implemented so that the sensor data size per transmission fits the block size. Encrypted sensor data from each sensor can be decrypted independently. One of the drawbacks of bitslice implementation has been the low number of applications where the encryption/decryption unit size is large, e.g., 2048-byte chunks. However, in this use case, one can simply collect encrypted sensor data from many sensors until the decryption unit size with no concern about the order, and then decrypt them by using bitslice implementation. The decryption key can be set block-by-block independently.

Our target. We choose PRESENT and Piccolo as each representative of lightweight block ciphers based on Substitution Permutation Networks and Feistel networks, respectively. Our implementations of PRESENT and Piccolo are run on three different Intel microarchitectures: Core (45-nm), Nehalem, and Sandy Bridge. Core and Nehalem support up to Streaming SIMD Extensions (SSE) 4.1 with 16 128-bit XMM registers, and Sandy Bridge newly supports Advanced Vector Extensions (AVX) as an extension of SSE. Major enhancements of AVX are supports for 256-bit YMM registers, 256-bit floating point instruction set, and 3-operand syntax, which is also used for legacy 128-bit SSE instructions (we call this 128-bit AVX). For example, 2-operand syntax instruction `pxor xmm1, xmm2` (`xmm1^=xmm2`) can be expressed in 3-operand syntax as `vpxor xmm1, xmm2, xmm3` (`xmm1=xmm2^xmm3`). Since a source operand of an instruction is not overwritten by the result, 3-operand syntax can reduce the cost of temporary data copy to another register and reduce code size. Unfortunately, 256-bit AVX does not support integer instructions operated on 256-bit YMM registers, so we use 128-bit AVX with 3-operand using XMM registers on Sandy Bridge. Legacy SSE instructions used in our implementation, such as logical (`pand`, `pandn`, `por`, `pxor`), data transfer (`movdqa`), shuffle (`pshufb`, `pshufd`), and unpack instructions (`punpckhbw` and its variants) are supported by the three architectures. 128-bit AVX instructions used on Sandy Bridge are `vpand`, `vpandn`, `vpor`, `vpxor`, `vpshufb`, `vpunpckhbw` and its variants. The latency of the register-to-register operations above is one cycle. The register-to-memory operations require more cycles depending on the data dependency, memory/cache mechanism, and characteristics of each microarchitecture.

Our implementation approach. Our implementation handles the number of parallel blocks smaller than the original bitslice implementation. This approach enables processing operations on only 16 XMM registers without frequent loading and storing of data between XMM registers and memory, and improves convenience as a cryptographic library tool. To explore the possibility of bitslice implementation of PRESENT and Piccolo, we study several cases for the number of blocks processed in parallel: 8-, 16-, and 32-parallelism for PRESENT and 16-parallelism for Piccolo. In Section 3 and 4, at the beginning we introduce some specific optimizations for each algorithm with legacy SSE instructions, and then optimize our implementations to reduce the number of instructions of the codes by using 128-bit AVX instructions on Sandy Bridge.

3 PRESENT

PRESENT [3] is a 64-bit block cipher supporting 80- and 128-bit keys. The S/P-network of PRESENT consists of addRoundKey, sBoxLayer and pLayer with 31 rounds as shown in Fig. 2. sBoxLayer consists of 16 parallel 4-bit S-boxes and pLayer permutes bit positions of the 64-bit data state. After the final round, the state is XORed with the round key for post-whitening and output as ciphertext. We denote the 64-bit block of PRESENT by 16 4-bit data $n_0, \cdots, n_{15}$. Let $n_i = n_{i,0}||n_{i,1}||n_{i,2}||n_{i,3}$ for $0 \leq i \leq 15$, where $n_{i,j}$ is the j-th bit of n_i.

Fig. 2. The S/P network for PRESENT

3.1 Bitsliced representation

Our bitsliced representations for 8-, 16-, and 32-block parallel implementations are shown in Fig. 3, Fig. 4, and Figs. 5 and 6, respectively. In this paper, we denote 16 128-bit XMM registers by $r[i], 0 \leq i \leq 15$. For l-block parallel implementation, l-bit data $\mathbf{n}_{i,j}$ in the figures means the bit collection of $n_{i,j}$ gathered from the same position of each l-block. The 4-bit slicing enables us to compute 4-bit S-box using bit-logical instructions in the same way as the original 1-bit slicing [2] and the 8-bit slicing for AES [12].

We use four XMM registers for 8-block parallel implementation, eight XMM registers for 16-block parallel implementation to store the 4-bit slicing of input data, and the remaining XMM registers as temporary registers for processing sBoxLayer and pLayer.

For 32-block parallel implementation, we handle two bitsliced representations with 16 XMM registers and switch the representations of intermediate data alternately in rounds to reduce the cost of pLayer processing, i.e., the processing can be skipped every other round. Figure 5 gives the initial bitsliced representation after performing a conversion algorithm. Since there are no XMM register for temporary use, we need to move data in a XMM register to memory in the process of sBoxLayer and pLayer.

Fig. 3. Bitsliced representation of PRESENT in 8-block parallel implementation

Fig. 4. Bitsliced representation of PRESENT in 16-block parallel implementation

Fig. 5. First bitsliced representation of PRESENT in 32-block parallel implementation

Fig. 6. Second bitsliced representation of PRESENT in 32-block parallel implementation

3.2 sBoxLayer

A smaller logical representation of S-box maximizes the advantage of bitslice implementation. One previous work reported that the logical representation of PRESENT S-box requires only 14 gates [4], in which four temporary registers were used and 3-operand logical instructions were assumed. We therefore use their logical representation for 8- and 16-block parallel implementations with 128-bit AVX on Sandy Bridge, and search for another logical representation using 2-operand instructions for the other implementations.

We took the same approach as Osvik [16] to search a software-oriented logical representation that consists of five operations `and`, `or`, `xor`, `not`, `mov` with only five registers (four registers for input and one register for temporary use). The instruction sequence found by our algorithm requires 20 instructions as below.

```
// Input:  r3, r2, r1, r0, tmp
// Output: r3, r2, r1, r0
 1. r2  ^= r1;   r3  ^= r1;
 2. tmp =  r2;   r2  &= r3;
 3. r1  ^= r2;   tmp ^= r0;
 4. r2  =  r1;   r1  &= tmp;
 5. r1  ^= r3;   tmp ^= r0;
 6. tmp |= r2;   r2  ^= r0;
 7. r2  ^= r1;   tmp ^= r3;
 8. r2  = ~r2;   r0  ^= tmp;
 9. r3  =  r2;   r2  &= r1;
10. r2  |= tmp;
11. r2  = ~r2;
```

Note that four registers `r3, r2, r1, r0` of input registers contain four input bits (`r3` contains the most significant bit).

3.3 pLayer

The original 1-bit slicing can compute bit-by-bit permutation like pLayer of PRESENT by only changing the order of registers with no cost. However our 4-bit slicing causes additional operations for processing pLayer in compensation for the decrease in the parallelism of bitslice implementations from 128 (size of XMM register) to 8, 16, and 32.

A combination of the shuffle byte instruction `pshufb` firstly introduced in Intel Supplemental SSE3 (SSSE3) and the unpack instructions for double-word `punpck(h/l)dq` and quad-word `punpck(h/l)qdq` realizes the pLayer processing. The notation `h` and `l` of `h/l` means high-order and low-order of 64-bit data in a 128-bit XMM register, respectively.

As the bitsliced representations for 8- and 16-block parallel implementation are almost same format, the implementation of pLayer for the 16-block requires the operations for the 8-block twice. We explain the case for the 8-block and then progress to the case for 32-block parallel implementation.

8-block parallel implementation. First of all, we perform `pshufb` on XMM register $r[0]$ containing $\mathbf{n}_{i,0}$ for $0 \leq i \leq 15$ in Fig. 3 as the following pattern.

$$r[0] : \mathbf{n}_{0,0}||\mathbf{n}_{4,0}||\mathbf{n}_{8,0}||\mathbf{n}_{12,0}||\mathbf{n}_{1,0}||\mathbf{n}_{5,0}||\cdots||\mathbf{n}_{10,0}||\mathbf{n}_{14,0}||\mathbf{n}_{3,0}||\mathbf{n}_{7,0}||\mathbf{n}_{11,0}||\mathbf{n}_{15,0}$$

Applying for the other registers $r[1], r[2]$, and $r[3]$ similarly, we perform the `punpckhdq` instruction on $r[0]$ and $r[1]$, which unpacks and interleaves the high-order double-word from $r[0]$ and $r[1]$ into $r[0]$. The subsequent `punpckhqdq` for $r[0]$ and $r[2]$, where $r[2]$ contains the result of `punpckhdq` for $r[2]$ and $r[3]$, can produce desired 128-bit data in register $r[0]$ as follows.

$$r[0] : \mathbf{n}_{0,0}||\mathbf{n}_{4,0}||\mathbf{n}_{8,0}||\mathbf{n}_{12,0}||\mathbf{n}_{0,1}||\mathbf{n}_{4,1}||\cdots||\mathbf{n}_{8,2}||\mathbf{n}_{12,2}||\mathbf{n}_{0,3}||\mathbf{n}_{4,3}||\mathbf{n}_{8,3}||\mathbf{n}_{12,3}$$

In the pLayer processing with legacy SSE instructions, we require 16 instructions, i.e., four `pshufb`, four `punpck(h/l)dq`, four `punpck(h/l)qdq`, and four `movdqa` for storing intermediate results. With an optimization using 128-bit AVX instructions `vpunpck(h/l)dq` and `vpunpck(h/l)qdq`, four `movdqa` become redundant, i.e., requiring 12 instructions in total.

32-block parallel implementation. As mentioned before, we manage two bitsliced representations for 32-block parallel implementation. These representations are constructed in such a way that the bit permutation of pLayer for the initial bitsliced representation as shown in Fig. 5 produces the other representation with only register renaming. Using the notation of the intial bitsliced representation, we can represent the updated bitsliced representation as the result of the pLayer process for the initial bitsliced representation as follows.

$$\begin{aligned}
r[0] &: \mathbf{n}_{0,0}||\mathbf{n}_{4,0}||\mathbf{n}_{8,0}||\mathbf{n}_{12,0}\\
r[4] &: \mathbf{n}_{1,0}||\mathbf{n}_{5,0}||\mathbf{n}_{9,0}||\mathbf{n}_{13,0}\\
r[8] &: \mathbf{n}_{2,0}||\mathbf{n}_{6,0}||\mathbf{n}_{10,0}||\mathbf{n}_{14,0}\\
r[12] &: \mathbf{n}_{3,0}||\mathbf{n}_{7,0}||\mathbf{n}_{11,0}||\mathbf{n}_{15,0}\\
&\vdots\\
r[3] &: \mathbf{n}_{0,3}||\mathbf{n}_{4,3}||\mathbf{n}_{8,3}||\mathbf{n}_{12,3}\\
r[7] &: \mathbf{n}_{1,3}||\mathbf{n}_{5,3}||\mathbf{n}_{9,3}||\mathbf{n}_{13,3}\\
r[11] &: \mathbf{n}_{2,3}||\mathbf{n}_{6,3}||\mathbf{n}_{10,3}||\mathbf{n}_{14,3}\\
r[15] &: \mathbf{n}_{3,3}||\mathbf{n}_{7,3}||\mathbf{n}_{11,3}||\mathbf{n}_{15,3}
\end{aligned}$$

The above corresponds to the second bitsliced representation as shown in Fig. 6. The pLayer processing in the next round for this representation requires an instruction sequence consisting of 16 `punpck(h/l)dq`, 16 `punpck(h/l)qdq` and 20 `movdqa` including four memory accesses for temporarily storing data twice and produces the initial bitsliced representation. Therefore the pLayer process can be computed every other round and requires 26 instructions on average. The 128-bit AVX can reduce the number of instructions from 52 to 36.

An additional operation for this trick to adjust the alignment of round keys is needed, unpacking round keys every other round in the key schedule.

4 Piccolo

Piccolo [20] is a lightweight 64-bit block cipher supporting 80-bit and 128-bit keys. Piccolo has a structure of a variant of 4-line generalized Feistel network (GFN) as shown in Fig. 7, and iterates 25 and 31 rounds for 80- and 128-bit keys, respectively. We denote a 64-bit block for Piccolo by four 16-bit words: W_0, W_1, W_2, W_3. Let $W_i = n_{4*i}||n_{4*i+1}||n_{4*i+2}||n_{4*i+3}$ for $0 \leq i \leq 3$, and let $n_j = n_{j,0}||n_{j,1}||n_{j,2}||n_{j,3}$ for $0 \leq j \leq 15$, where n_j is 4-bit data and $n_{j,k}$ is the k-th bit of n_j.

Fig. 7. The structure of Piccolo

Fig. 8. Round permutation RP

4.1 Bitsliced Representation

Figure 9 shows our bitsliced representation for 16-block parallel implementation of Piccolo. 16-bit data $\mathbf{n}_{i,j}$ in the figure means the bit collection of $n_{i,j}$ gathered from a same position of each 16-block for $0 \leq i \leq 15$ and $0 \leq j \leq 3$.

Since the 2-line out of the 4-line GFN is processed and the other lines pass through in one round, the data for the former and latter should be stored separately, assigning each for different registers. Then, two F-functions used in the GFN are the same, so we can pack eight 16-bit data $\mathbf{n}_{i,j}$ corresponding to the 2-line data (e.g., $W_0\&W_2$ or $W_1\&W_3$) on same XMM registers with the 4-bit slicing as our implementation of PRESENT.

The number of XMM registers for storing 4-bit slicing of input data is eight. Four XMM registers of the renaming eight XMM registers can be used for storing the data passed through during processing F-functions and the other four XMM registers can be used as temporary registers for processing F-functions.

If we assign the data for a line of the 4-line GFN on a XMM register for 32-block parallel implementation of Piccolo, we would need full 16 XMM registers to store 4-bit slicing of input data. It leads to more memory access than the case of PRESENT for storing the data temporarily. Therefore we think 16-block parallel implementation of Piccolo using 128-bit XMM registers is optimal parallelism.

Fig. 9. Bitsliced representation of Piccolo in 16-block parallel implementation

4.2 F-function

The F-function consists of two S-box layers and a diffusion matrix (see Fig. 10).

Fig. 10. F-function

Fig. 11. S-box S

S-box layer. The S-box layer consists of four 4-bit bijective S-boxes S represented by the logic circuit shown in Fig. 11. A software instruction sequence of the S-box can be manually obtained from the logic circuit, which requires 15 instructions with a temporary register. We searched for a smaller instruction sequence in the similar way to the PRESENT S-box and found the following one with 13 instructions in six cycles, assuming that up to three independent instructions are issued per cycle.

```
// Input:  r3, r2, r1, r0, tmp
// Output: r0, r1, r2, r3
1. tmp =  r1;   r1  |= r2;   r3  = ~r3;
2. r0  ^= r2;   r1  ^= r3;   r3  |= r2;
3. r0  ^= r3;   r3  =  r1;
4. r3  |= r0;
5. r3  ^= tmp;  tmp |= r0;
6. r2  ^= tmp;  r3  = ~r3;
```

The notation is the same as the instruction sequence of the PRESENT S-box.

Diffusion Matrix. The following multiplication between the constant 4×4 diffusion matrix M and four 4-bit data x_0, x_1, x_2, x_3 over GF(2^4) defined by an irreducible polynomial $x^4 + x + 1$ outputs four 4-bit data y_0, y_1, y_2, y_3.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Let $x_i = x_{i,0}||x_{i,1}||x_{i,2}||x_{i,3}$ for $0 \leq i \leq 3$. Since diffusion matrix M is cyclic, 4-bit data y_i can be expressed as $y_i = 2 \cdot x_i \oplus 3 \cdot x_{i+1} \oplus x_{i+2} \oplus x_{i+3}$ where index is calculated by modulo 4. Representing each bit of the products $2 \cdot x_i$ and $3 \cdot x_i$ given in Table 1, all bits of $y_i = y_{i,0}||y_{i,1}||y_{i,2}||y_{i,3}$ are obtained as follows.

$$\begin{aligned} y_{i,0} &= x_{i,1} \oplus x_{i+1,0} \oplus x_{i+1,1} \oplus x_{i+2,0} \oplus x_{i+3,0} \\ y_{i,1} &= x_{i,2} \oplus x_{i+1,1} \oplus x_{i+1,2} \oplus x_{i+2,1} \oplus x_{i+3,1} \\ y_{i,2} &= x_{i,0} \oplus x_{i,3} \oplus x_{i+1,0} \oplus x_{i+1,2} \oplus x_{i+1,3} \oplus x_{i+2,2} \oplus x_{i+3,2} \\ y_{i,3} &= x_{i,0} \oplus x_{i+1,0} \oplus x_{i+1,3} \oplus x_{i+2,3} \oplus x_{i+3,3} \end{aligned}$$

As each 128-bit XMM register contains eight 16-bit data corresponding the 2-line data, it is possible to perform two matrix calculations simultaneously, utilizing rotation of the data on the upper and lower 64-bit data in a 128-bit XMM register. The 16-bit left rotation rot_{16} on a XMM register with `pshufb` instruction is defined as follows.

$$rot_{16} : [w_0||w_1||w_2||w_3||w_4||w_5||w_6||w_7] \mapsto [w_1||w_2||w_3||w_0||w_5||w_6||w_7||w_4]$$

Note that w_i is a 16-bit data and 32-bit left rotation rot_{32} and 48-bit left rotation rot_{48} can be defined in the same way.

Table 1. Multiplication 4-bit data x_i by 2 and 3 over GF(2^4) defined by $x^4 + x + 1$

x_i	$x_{i,0}$	$x_{i,1}$	$x_{i,2}$	$x_{i,3}$
$2 \cdot x_i$	$x_{i,1}$	$x_{i,2}$	$x_{i,0} \oplus x_{i,3}$	$x_{i,0}$
$3 \cdot x_i$	$x_{i,0} \oplus x_{i,1}$	$x_{i,1} \oplus x_{i,2}$	$x_{i,0} \oplus x_{i,2} \oplus x_{i,3}$	$x_{i,0} \oplus x_{i,3}$

We can compute the following updated four XMM registers $r[i]$ for $0 \leq i \leq 3$ with 25 instructions including eight `pshufb`, 13 `pxor` and four `movdqa` instructions, using four temporary registers.

$$r[0] \leftarrow r[1] \oplus rot_{16}(r[1]) \oplus rot_{16}(r[0]) \oplus rot_{32}(r[0] \oplus rot_{16}(r[0]))$$
$$r[1] \leftarrow r[2] \oplus rot_{16}(r[2]) \oplus rot_{16}(r[1]) \oplus rot_{32}(r[1] \oplus rot_{16}(r[1]))$$
$$r[2] \leftarrow r[0] \oplus rot_{16}(r[0]) \oplus r[3] \oplus rot_{16}(r[3]) \oplus rot_{16}(r[2]) \oplus rot_{32}(r[2] \oplus rot_{16}(r[2]))$$
$$r[3] \leftarrow r[0] \oplus rot_{16}(r[0]) \oplus rot_{16}(r[3]) \oplus rot_{32}(r[3] \oplus rot_{16}(r[3]))$$

Note that translating $rot_{48} = rot_{32} \circ rot_{16}$ saves the number of `pshufb`.

4.3 Round Permutation

The round permutation (RP) permutes eight 8-bit data over 64-bit data as shown in Fig. 8. A simple implementation of RP for a XMM register holding j-th bit of 4-bit data n_i permutes four 32-bit data (e.g., $\mathbf{n}_{i,j}, \mathbf{n}_{i+1,j}$) in a 128-bit register by using double-word shuffle instruction `pshufd` as follows.

$$rp_0 : [w_0||w_1||w_2||w_3||w_4||w_5||w_6||w_7] \mapsto [w_4||w_5||w_2||w_3||w_0||w_1||w_6||w_7]$$
$$rp_1 : [w_0||w_1||w_2||w_3||w_4||w_5||w_6||w_7] \mapsto [w_0||w_1||w_6||w_7||w_4||w_5||w_2||w_3]$$

Note that we perform rp_0 and rp_1 on four XMM regsiters holding the data for $W_0\&W_2$ and $W_1\&W_3$, respectively. It requires only two `pshufd` instructions per bit, or eight in total per one round. Before proceeding to the next round, we need renaming four XMM regsiters holding the data for $W_0\&W_2$ and $W_1\&W_3$.

Remove round permutation. We describe the implementation to remove RP changing the calculation of diffusion matrix M and the position of round keys with no cost in the data processing. This modification can reduce 8*(the number of rounds) instructions compared to the above implementation of RP.

Since register renaming can only switch the positions of the 2-line data, the removing RP causes the misalignment in byte position on XMM registers to effect input-output of diffusion matrix and subsequent xor with round keys and data in the previous round. The byte positions from Round 1 to 5 in normal 64-bit block with/without RP before the round process are as follows.

	byte position with RP	byte position without RP
Round 1:	$[b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]$	$[b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]$
Round 2:	$[b_2, b_7, b_4, b_1, b_6, b_3, b_0, b_5]$	$[b_2, \boldsymbol{b}_3, \boldsymbol{b}_0, b_1, b_6, \boldsymbol{b}_7, \boldsymbol{b}_4, b_5]$
Round 3:	$[b_4, b_5, b_6, b_7, b_0, b_1, b_2, b_3]$	$[\boldsymbol{b}_0, \boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3, \boldsymbol{b}_4, \boldsymbol{b}_5, \boldsymbol{b}_6, \boldsymbol{b}_7]$
Round 4:	$[b_6, b_3, b_0, b_5, b_2, b_7, b_4, b_1]$	$[\boldsymbol{b}_2, b_3, b_0, \boldsymbol{b}_1, \boldsymbol{b}_6, b_7, b_4, \boldsymbol{b}_5]$
Round 5:	$[b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]$	$[b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]$

Note that b_i is 8-bit data corresponding to a pair of 4-bit data n_{2*i} and n_{2*i+1} for $0 \leq i \leq 7$, and the misalignment of byte position is emphasized by bold phase. The above shows that the misalignment disappears in four rounds.

In Round 2, two 8-bit data b_3, b_0 switch positions with two 8-bit data b_7, b_4 for the input data $(b_2, \boldsymbol{b}_3, b_6, \boldsymbol{b}_7)$ and output data $(\boldsymbol{b}_0, b_1, \boldsymbol{b}_4, b_5)$ of two F-functions. Utilizing the shuffle operations rp_1, rp_0 to cancel the effect of each misalignment, we introduce $\mathit{shf}_0 = rp_0 \circ rp_1$ and replace rot_{16}, rot_{32} in the original diffusion matrix with $\mathit{shf}_{16} = rp_0 \circ rot_{16} \circ rp_1$, $\mathit{shf}_{32} = rp_0 \circ rot_{32} \circ rp_0$ as below.

$$\begin{aligned}
\mathit{shf}_0 &: [x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7] \mapsto [x_4||x_5||x_6||x_7||x_0||x_1||x_2||x_3] \\
\mathit{shf}_{16} &: [x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7] \mapsto [x_5||x_2||x_7||x_0||x_1||x_6||x_3||x_4] \\
\mathit{shf}_{32} &: [x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7] \mapsto [x_6||x_7||x_4||x_5||x_2||x_3||x_0||x_1]
\end{aligned}$$

The new representation for calculating diffusion matrix can be expressed with 25 instructions including eight `pshufb`, four `pshufd`, and 13 `pxor` as follows.

$$\begin{aligned}
r[0] &\leftarrow \mathit{shf}_0(r[1]) \oplus \mathit{shf}_{16}(r[1]) \oplus \mathit{shf}_{16}(r[0]) \oplus \mathit{shf}_{32}(\mathit{shf}_0(r[0]) \oplus \mathit{shf}_{16}(r[0])) \\
r[1] &\leftarrow \mathit{shf}_0(r[2]) \oplus \mathit{shf}_{16}(r[2]) \oplus \mathit{shf}_{16}(r[1]) \oplus \mathit{shf}_{32}(\mathit{shf}_0(r[1]) \oplus \mathit{shf}_{16}(r[1])) \\
r[2] &\leftarrow \mathit{shf}_0(r[0]) \oplus \mathit{shf}_{16}(r[0]) \oplus \mathit{shf}_{16}(r[2]) \oplus \mathit{shf}_0(r[3]) \oplus \mathit{shf}_{16}(r[3]) \\
&\qquad \oplus \mathit{shf}_{32}(\mathit{shf}_0(r[2]) \oplus \mathit{shf}_{16}(r[2])) \\
r[3] &\leftarrow \mathit{shf}_0(r[0]) \oplus \mathit{shf}_{16}(r[0]) \oplus \mathit{shf}_{16}(r[3]) \oplus \mathit{shf}_{32}(\mathit{shf}_0(r[3]) \oplus \mathit{shf}_{16}(r[3]))
\end{aligned}$$

We omit `movdqa` in the original diffusion matrix, utilizing `pshufd` natively supporting 3-operand. In Round 3 for the input data $(\boldsymbol{b}_0, \boldsymbol{b}_1, \boldsymbol{b}_4, \boldsymbol{b}_5)$ of two F-functions, b_0, b_1 switches positions with b_4, b_5, but we can use the original representation owing to the calculation of two diffusion matrices independently. Since the misalignment of output data $(\boldsymbol{b}_2, \boldsymbol{b}_3, \boldsymbol{b}_6, \boldsymbol{b}_7)$ is the same for the input data, no operations are needed. Round 4 can use the same representation in Round 2.

Therefore, we alternately call the original diffusion matrix and modified one, and adjust the data alignment for the round keys in the key schedule. With 128-bit AVX, the modified representation of diffusion matrix in Round 2 requires four more instructions compared to the original one, so the performance improvement remains about three fourths of the case with legacy SSE instructions.

5 Performance

This section summarizes the instruction counts for PRESENT and Piccolo, and shows the evaluation results of our implementations on three different computers given in Table 2.

Table 2. Computers used for benchmarking

Processor	Intel Xeon E5410	Intel Core i7 870	Intel Xeon E3-1280
Microarchitecture	Core	Nehalem	Sandy Bridge
Clock Speed	2.33 GHz	2.93 GHz	3.5 GHz
RAM	8 GB	16 GB	16 GB
OS	Linux 2.6.16.60 x86_64	Linux 3.1.10 x86_64	Linux 2.6.37.6 x86_64

Table 3. Instruction count for PRESENT and Piccolo with Legacy SSE instructions

	logical instr.	mov	shuffle	unpack	mov (mem)	xor (mem)	per round	TOTAL 80-bit	TOTAL 128-bit
PRESENT (8-block parallel)							40	**1444**	
addRoundKey	-	-	-	-	-	4	4	128	
sBoxLayer	17	3	-	-	-	-	20	620	
pLayer	-	4	4	8	-	-	16	496	
conversion	154	28	12	8	8	-	-	200	
PRESENT (16-block parallel)							80	**2720**	
addRoundKey	-	-	-	-	-	8	8	256	
sBoxLayer	34	6	-	-	-	-	40	1240	
pLayer	-	8	8	16	-	-	32	992	
conversion	154	32	24	16	16	-	-	232	
PRESENT (32-block parallel)							126*	**4446**	
addRoundKey	-	-	-	-	-	16	16	512	
sBoxLayer	68	12	-	-	4	-	84	2604	
pLayer	-	0/16	-	0/32	0/4	-	0/52	780	
conversion	288	78	82	64	38	-	-	550	
Piccolo (16-block parallel)							63	**1815**	**2193**
diffusion matrix	13	4/0	8/12	-	-	-	25	625	775
S-box	22	4	-	-	-	-	26	650	806
addRoundKey	4	4	-	-	-	4	12	300	372
addWhiteningKey	-	-	-	-	-	8	-	8	
conversion	154	32	24	16	16	-	-	232	

Table 3 presents the total number of instructions for PRESENT and Piccolo with the legacy SSE instruction set. The notations "logical instr." and "(mem)" in the table mean logical instructions including shift operation and instructions with memory, respectively. For the 32-block parallel implementation of PRESENT, "*" means the number of instructions per round on average. The "diffusion matrix" in Piccolo shows both the number of instructions for calculating the original diffusion matrix (left) and that for modified one (right). The "conversion" includes not just conversion process that converts input data to the bitsliced representation and reverses it to output data, but also loading input data and storing output data. Our conversion algorithm utilizes a part of the assembly code published by Käsper and Schwabe [12], which includes 84 instructions to convert eight 128-bit blocks on eight XMM registers to the bitsliced representation of 8-bit slicing on eight XMM registers with one temporary XMM register. We added a few shuffle and unpack instructions in this code to obtain desired bitsliced format.

We optimized our implementation with 128-bit AVX instructions. Owing to 3-operand syntax, the number of `mov` instructions in the table is zero except for register-to-memory operations. Furthermore we can use smaller instruction sequence of PRESENT S-box with 14 instructions in 8- and 16-block parallelism.

Table 4. Performance of PRESENT and Piccolo with 80-bit and 128-bit keys

Algorithm	PRESENT-80/128			Piccolo-80	Piccolo-128
Number of parallel blocks	8	16	32	16	
Xeon E3-1280 (Sandy Bridge)					
Cycles/byte	8.46	6.52	4.73	4.57	5.52
Instructions/cycle	2.04	2.48	3.10	2.61	2.61
Core i7 870 (Nehalem)					
Cycles/byte	10.88	7.26	5.79	5.69	6.80
Instructions/cycle	2.07	2.93	3.00	2.49	2.52
Xeon E5410 (Core)					
Cycles/byte	13.55	10.98	7.55	6.85	8.23
Instructions/cycle	1.67	1.93	2.30	2.07	2.08

The numbers of instructions for 8-, 16-, and 32-block parallel implementation of PRESENT with 128-bit AVX are **1106**, **2068**, and **3752**, respectively. The numbers of instructions for 16-block parallel implementation of Piccolo with 128-bit AVX are **1531** and **1849** for 80- and 128-bit keys, respectively.

Table 4 gives evaluation results. We measured the average cycles of encryptions for 1024KB random data and did not include the cost of the key schedule, which was regarded as negligible cost in our evaluation. Since the number of rounds of PRESENT is 31 for both 80- and 128-bit keys, the results of PRESENT-80 and -128 are exactly the same. The result on Xeon E5410 shows the performance of optimized code with 128-bit AVX.

For comparison, only a few software implementation results of ultra-lightweight block ciphers on general-purpose processors have been reported. A table-based implementation of LED [8] with 64- and 128-bit keys needs 57 and 86 cycles on Core i7 Q720 (1.60 GHz). Suzaki et al. [21] showed that TWINE encryption achieved 11.0 cycles/byte on Core i7 2600S (2.8 GHz, Sandy Bridge), so our implementations of PRESENT and Piccolo deliver superior performance compared with previous results and indicate an attractive option for software implementation for lightweight block ciphers on general-purpose processors.

As far as we know, besides hardware efficiency, Piccolo-80 achieves the fastest software implementation among existing 64-bit block ciphers in our implementation. Moreover, since Piccolo adopts a permutation based key schedule, which is lighter than the S-box based key schedule of PRESENT, Piccolo may have some advantage even for short message encryption. On the other hand, there are some stream ciphers with small hardware and fast software performance. For example, the public eBASC benchmarks report that TRIVIUM achieves 3.69 cycles/byte and SNOW 2.0 achieves 4.03 cycles/byte on a Nehalem CPU (dragon).

For further optimization, 256-bit AVX accelerates the performance of our bit-slice implementation with low parallelism using AVX2 instruction set introduced in Haswell microarchitecture which will be released in 2013. An optimization for instruction sequences of S-box assuming both 3-operand instructions and issuing three independent instructions remains the matter of research.

6 Conclusion

This paper showed the great potential of lightweight cryptography in fast and timing-attack resistant software implementations in cloud computing by exploiting bitslice implementation. This was demonstrated by bitslice implementations of the PRESENT and Piccolo lightweight block ciphers. In particular, PRESENT-80/128 achieved 5.79 cycles/byte and Piccolo-80 achieved 5.69 cycles/byte on an Intel Core i7 processor, which is faster than the AES speed record in bitslice implementation on the same microarchitecture. We demonstrated bitslice implementation of only two lightweight block ciphers, but other lightweight block ciphers as well as other lightweight cryptographic primitives such as hash functions are worth implementing. We hope that lightweight cryptography will be used not only for constrained devices, but also for cloud computing.

Acknowledgments

The authors appreciate Kazuya Kamio and Kyoji Shibutani for useful comments and suggestions.

References

1. Aumasson, J.-P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A Lightweight Hash. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010, LNCS, vol. 6225, pp. 1–15. Springer (2010)
2. Biham E.: A Fast New DES Implementation in Software. In: Biham, E. (ed.) FSE 1997, LNCS, vol. 1267, pp. 260–272. Springer (1997)
3. Bogdanov, A., Knudsen L., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007, LNCS, vol. 4727, pp. 450–466. Springer (2007)
4. Courtois, N. T., Hulme, D., Mourouzis, T.: Solving Circuit Optimization Problems in Cryptography and Cryptanalysis. Cryptology ePrint Archive, Report 2011/475 (2011), `http://eprint.iacr.org/2011/475`
5. De Cannière, C., Dunkelman, O., Knezevic, M.: KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009, LNCS, vol. 5747, pp. 272–288. Springer (2009)
6. Grabher, P., Großschädl, J., Page, D.: Light-Weight Instruction Set Extensions for Bit-Sliced Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008, LNCS, vol. 5154, pp. 331–345. Springer (2008)
7. Guo, J., Peyrin T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In: Rogaway, P. (ed.) CRYPTO 2011, LNCS, vol. 6841, pp. 222–239. Springer (2011)
8. Guo, J., Peyrin T., Poschmann, A., Robshaw, M.: The LED Block Cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011, LNCS, vol. 6917, pp. 326–341. Springer (2011)
9. Guo, J., Peyrin T., Poschmann, A., Robshaw, M.: The LED Block Cipher. Presentation slide at CHES 2011. (2011), `http://www.iacr.org/workshops/ches/ches2011/presentations/Session%207/CHES2011_Session7_2.pdf`

10. Intel 64 and IA-32 Architectures Optimization Reference Manual, http://www.intel.com/
11. ISO/IEC 29192-2:2012, Information technology – Security techniques – Lightweight cryptography – Part 2: Block ciphers (2012)
12. Käsper, E., Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009, LNCS, vol. 5747, pp. 1–17. Springer (2009)
13. Könighofer, R.: A Fast and Cache-Timing Resistant Implementation of the AES. In: Malkin, T. (ed.) CT-RSA 2008, LNCS, vol. 4964, pp. 187–202. Springer (2008)
14. Lee, E.: Cyber Physical Systems: Design Challenges. EECS Department, University of California, Berkeley (2008)
15. Matsui, M., Nakajima, J.: On the Power of Bitslice Implementation on Intel Core2 Processor. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007, LNCS, vol. 4727, pp. 121–134. Springer (2007)
16. Osvik, D. A.: Speeding up Serpent. AES Candidate Conference, pp. 317–329. (2000)
17. Poschmann, A.: Lightweight Cryptography – Cryptographic Engineering for a Pervasive World. Cryptology ePrint Archive, Report 2009/516 (2009), http://eprint.iacr.org/2009/516
18. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Al-Shaer, E., Jha, S., Keromytis, A. (eds.) ACM Conference on Computer and Communications Security, pp. 199–212. ACM (2009)
19. Shamir, A: SQUASH – A New MAC with Provable Security Properties for Highly Constrained Devices Such as RFID Tags. In: Nyberg, K. (ed.) FSE 2008, LNCS, vol. 5086, pp. 144–157. Springer (2008)
20. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An Ultra-Lightweight Blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011, LNCS, vol. 6917, pp. 342–357. Springer (2011)
21. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.,: TWINE: A Lightweight, Versatile Block Cipher. In: Leander, G., Standaert, F., (eds.) ECRYPT Workshop on Lightweight Cryptography 2011, pp. 146–169. (2011)