FACE: Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data

Jin Hyung Park and Dong Hoon Lee

Center for Information Security Technologies,
Korea University, Seoul, Republic of Korea
jhpark.embedsec.korea@gmail.com,donghlee@korea.ac.kr

Abstract. The Advanced Encryption Standard (AES) algorithm and Counter (CTR) mode are used for numerous services as an encryption technique that provides confidentiality. Even though the AES with counter (AES CTR) mode has an advantage in that it can process multiple data blocks in parallel, its implementation should also be observed to reduce the computational burden of current services.
In this paper, we propose an implementation method called FACE that can improve the performance of the AES CTR mode. The proposed method is based on five caches of frequently occurring intermediate values, so that it reduces the number of unnecessary computations. Our method can be employed in any AES CTR implementation, regardless of the platform, environment, or implementation method. There are two known AES implementation techniques, namely, counter-mode caching and bitslicing. FACE extends counter-mode caching in order to optimize the previous result and to maximize the scope of caching. We show that FACE can be applied efficiently to various implementations (table-based, bitsliced, and AES-NI-based). In particular, this is the first attempt to combine our extended counter-mode caching with bitsliced implementations of AES, and is also the first to apply counter-mode caching up to the round transformations of AES-NI implementation. To prove the efficiency of our proposed method, we conduct a performance evaluation in various environments, which we then compare with the previous fastest results. Our bitsliced FACE needs 6.41 cycles/byte on an Intel Core 2, and AES-NI-based FACE records 0.44 cycles/byte on an Intel Core i7.

Keywords: AES · counter mode · efficient software implementation · AES bitslicing · AES-NI

1 Introduction

The number of Internet users has increased rapidly with significant improvements in network technologies and services such as content delivery and VoIP have also emerged in response to demand. Because these Internet services are commonly based on usage-pricing models, service providers should consider a way to protect their service assets from illegal usage. Although there are several technologies for protecting either assets or information, most of these are based on providing confidentiality for their contents. On the other hand, the privacy of users should also be protected while they access the Internet. As a result, security services such as SSL are now widely adopted in various environments. The above-mentioned issues can be resolved using a cryptographic algorithm (e.g., AES, DES) for data confidentiality. However, adopting a cryptographic algorithm for current services is burdensome because it requires additional computational resources. Therefore,

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925,
Vol. 2018, No. 3, pp. 469–499
DOI:10.13154/tches.v2018.i3.469-499

cryptographic algorithms must guarantee proven security and efficiency when employed in practical environments.

AES [NIS01a] and CTR mode [NIS01b] are used for numerous services (e.g., OMA DRM, VoIP, IPTV) as an encryption technique that preserves confidentiality. The AES CTR mode is not only operated as a standalone technique, but is also incorporated within authenticated encryption schemes, such as the AES GCM [NIS07] and AES CCM [NIS04]. Thus, optimizing the AES CTR mode results in improved performance not only for the AES CTR, but also the AES GCM and the AES CCM. While researches on improving the throughput of AES are ongoing, Intel has announced a set of instructions (AES-NI [Gue10]) that accelerate AES computations with dedicated hardware support. ARM also presents a set of instructions for accelerating AES on ARMv8 as crypto extension. However, AES-NI and Crypto Extension can be used only with specific processors. Many other processors (non-Intel, non-AMD, and pre-ARMv8-based) used in embedded devices or lightweight IoT devices do not yet support these instructions. This is the reason why enhancing the AES efficiency in software remains an important issue.

This study focuses on maximizing the efficiency of the AES algorithm using the counter mode. The AES CTR mode is one of the most widely used cryptographic algorithms for confidentiality, and is usually employed in seamless real-time services. The Open Mobile Appliance (OMA) Digital Right Management (DRM) v2.0 includes the AES CTR mode in PDCF format [All08] to protect streaming content such as music or video on mobile devices. Moreover, IPTV and VoIP services are high-profile Internet services that are usually based on the Secure Real-time Transfer Protocol (RTP) to protect data confidentiality. Even though Secure RTP improves security compared to previous versions of RTP, IETF has also selected the AES CTR mode with consideration for efficiency [BMN+04]. In general, AES CTR mode is widely adopted in many current applications for the following reasons. First, its security is proven in [McG02]. In addition, it has many advantages with respect to efficiency over AES with other operation modes, such as CBC. For example, the AES CTR mode can be processed in parallel, regardless of encryption or decryption. Moreover, it does not require the implementation of the AES decryption algorithm. When the services are deployed in lightweight devices that have relatively limited computational capabilities, it is even more important to conserve computation resources.

In this paper, we present an efficient implementation method for the AES CTR mode, called FACE (Fast AES CTR mode Encryption). FACE can be applied to existing implementation, regardless of platforms and implementation methods. The motive of our method can be summarized as follows. In the AES CTR mode, while increasing the number of blocks, the Initial Vector (`IV`) or counter that is used as an input value adds 1 to its least significant bit. The AES CTR mode encrypts sequentially increased `IV` rather than plaintext sequences. The output is XORed with the plaintext to produce a final ciphertext. We note that there are only small changes in each of the input blocks. The AES algorithm spreads the small variation in the input over the entire output by iterating its round transformation. This means that the front-located rounds operate a many-overlapped input in the AES CTR mode. If these overlapped values are cached and reused properly, the AES CTR mode can guarantee enhanced performance. Our proposed method, FACE, maximizes the scope for reuse in the AES CTR mode.

The contributions of our work are as follows.

1. We propose an efficient implementation technique for the CTR mode of AES. The proposed technique (FACE) extends the counter-mode caching that was first presented in [BS08] with credit to [Wu07]. Previous counter-mode caching technique only covered partial data of round transformation. Therefore, in order to show its efficiency, it has been applied only to table-based implementation. However, FACE can be employed in any AES CTR implementation, regardless of the platform, environment, or implementation method, as this technique can cover a round transformation

entirely.

2. We show that FACE can be applied efficiently to existing implementation methods (e.g., table-based, bitsliced, and AES-NI-based implementations). In particular, our work is the first to combine counter-mode caching with bitsliced implementations of AES, and is also the first to apply counter-mode caching up to the round transformations of AES-NI implementation. Table-based FACE needs 12 instructions up to round 2, whereas the existing implementation [Pro] needs 128 instructions. Further, bitsliced FACE requires 74 instructions up to round 2, whereas [KS09] requires 618 instructions. AES-NI-based FACE requires 1 intrinsic instruction, 1 memory reference, and 1 arithmetic instruction up to round 2, whereas [Lib] requires 3 intrinsic instructions and 3 memory references. According to the Intel instruction latency and throughput [Cor18], a briefly calculated throughput (required cycle) for AES-NI-based FACE is 0.83 up to round 2, whereas for [Lib] it is 3.08.

3. Bitsliced FACE records 6.41 cycles/byte on an Intel Core 2 Q9550, and AES-NI-based FACE records 0.44 cycles/byte on an Intel Core i7 8700K. Our experimental results are recorded as the highest throughput ever achieved.

The rest of this paper is organized as follows. In section 2, we show related works involving attempts to enhance the performance of AES. In section 3, we briefly describe the AES and CTR mode of operation, as well as the relevant notations. In section 4, we present the techniques of FACE, which utilize repetitive data. In section 5, we explain our implementation and give the experimental results of FACE. Then, we compare our results to those of other software implementations. Section 6 discusses the possibility of cache-timing attacks. Finally, we conclude this paper in section 7.

2 Related Work

Attempts to improve the efficiency of the AES algorithm can be divided into two categories. One is to improve the implementation method of the hardware architecture, and another is to reduce the logic at the software level. The hardware approach has a limited advantage though, as it is comparatively less applicable than software.

Morioka and Satoh proposed AES implementation [MS04] that applies T-box, which is a combination of AES transformations (`SubBytes`, `ShiftRows` and `MixColumns`) [DR13]. Rouvroy et al. [RSQL04] suggest a design that combines the key schedule part and the data path part in a Xilinx FPGA. Sagib et al. [SRHDP03] propose a sequential architecture and pipeline architecture in FPGA, and Charot et al. [CYW03] implement a single round as a module in the Altera single-chip FPGA, enabling the degree of pipelining to be determined flexibly. Although there have been some efforts to improve the algorithm, most of them were based on a hardware implementation using FPGA. By applying loop unrolling and pipelining, several FPGA-based AES implementations have achieved a high throughput [GCVRSPGP10][QSH+09]. However, these hardware-based techniques have some restrictions, i.e. although hardware implementations commonly offer a high throughput compared to software designs, it is difficult to update built-in cryptographic modules—when the improved implementation were announced, if we want to apply this announced method, the update of implementation may not be as easy as software is—and these efforts cannot be applied to application software.

Another approach to enhancing the efficiency of AES can be found in improving the implementation logic in software. A widely known software implementation method is S-Box with pre-computation, as proposed by the authors of AES. It combines `SubBytes`, `ShiftRows`, and `MixColumns`, which are parts of the round operation of AES, and generates a pre-computation look-up table. Consequently, these three transformations can be replaced

by a single lookup operation[1]. Bertoni et al. [BBF+02] report implementations for smaller CPUs by modifying the `MixColumns` transformation. Matsui and Fukuda [MF05] propose an efficient implementation that covers the Pentium III and Pentium 4. In addition, the most notable work is the AES implementation that uses the bitsliced method.

Matsui and Nakajima propose a bitslice AES implementation on an Intel Core 2, which is faster than any previous implementation [MN07]. The reported throughput of 9.2 cycles per byte is achieved only for a data chunk longer than 2,048 bytes. The bitsliced method works up to an enhanced performance as high as 7.59 cycles per byte on an Intel Core 2 Q9550 [KS09]. However, these works are not scalable because they can only operate on designated CPU architecture. Liu and Bass propose a parallel AES implementation [LB13] that achieves 70 cycles per block, although it requires a 164-core environment, which is not considered practical. On the other hand, techniques that optimize the number of calculations using the pre-computation method can be applied on various platforms regardless of their CPU architectures.

Bernstein and Schwabe [BS08] present a technique that improves the efficiency of the AES CTR mode, with credit to [Wu07]. Because of the property of the CTR mode, 15 bytes in the counter value of 16 bytes are maintained while processing 256 blocks, but only one byte changes. In [BS08], the calculation result is saved with the exception of the part influenced by the changed byte. The saved results are then reused while processing 256 blocks. This technique was first announced by Hongjun Wu, and can be found in the Crypto++ module [Lib] and in an open-source version of AES [Wu07]. In this paper, we maximize the reuse of intermediate values in the AES CTR mode. Our method improves [BS08, Lib, Wu07] and proposes novel techniques.

3 Preliminaries

In this section, we give a brief description of AES and CTR mode, as well as the relevant notation.

3.1 Description of AES and CTR mode

3.1.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES)[NIS01a] is a symmetric key block cipher announced by the National Institute of Standards and Technology (NIST) to supersede DES. It has a fixed block size of 128 bits and supports key sizes of 128, 192, or 256 bits. In accordance with the key sizes, the AES algorithm is categorized into three types: AES-128, AES-192, and AES-256. Internally, AES uses the SPN structure, and repetitively performs a round function. The number of rounds is 10 for AES-128, 12 for AES-192, and 14 for AES-256. The round function is composed of four types of transformations: `SubBytes, ShiftRows, MixColumns`, and `AddRoundKey`. Each transformation operates on *State*, which is treated as a 4×4 matrix of bytes. The following briefly describes the four types of transformations of the AES round function.

`SubBytes` This operation substitutes one byte with another byte according to the S-Box table in the AES algorithm. In `SubBytes`, a byte calculation is not affected by other input bytes because the substitution deals with single bytes independently. Further, the same S-Box input always produces the same output, regardless of its position and rounds.

`ShiftRows` This transformation circularly transposes rows of *State* matrix from right to left. The amount of transposition is relevant to the row position. There are four rows in *State*, and the first row remains fixed. The bytes in the second row circularly change

[1] *Some XOR operations still remain.*

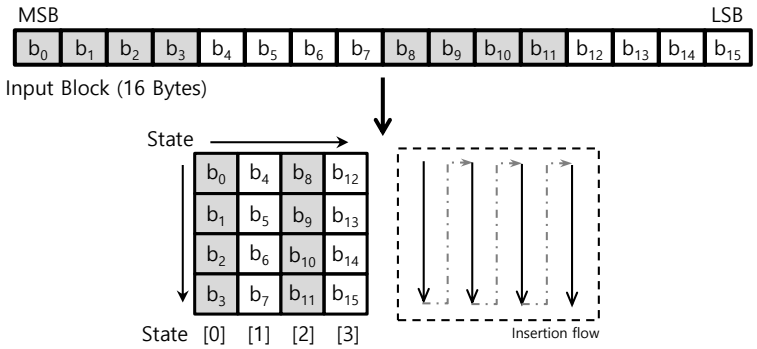

Figure 1: AES Block-to-State Transformation

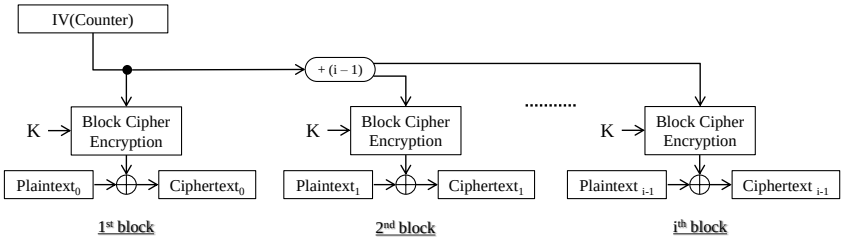

Figure 2: CTR Mode Encryption

their positions once, and the third and last rows change positions two and three times, respectively.

`MixColumns` While `ShiftRows` transforms bytes row-by-row, the `MixColumns` transformation is performed column-by-column. This operation combines the four bytes in each column. Each column of *State* matrix is changed into a new column using matrix multiplication with a constant matrix.

`AddRoundKey` This transformation simply XORs a given *State* with round keys. Each byte of *State* is XORed with the corresponding byte of the round key.

3.1.2 AES Block-to-State Transformation

To manipulate the internal input and output data, the AES algorithm uses *State* to form a series of data into a matrix. The AES data block can be expressed as a matrix of 1×16 bytes. *State* is 128 bits, but is expressed as a 4×4 matrix. At the beginning of the algorithm, the bytes in the input data block are inserted column-by-column into *State*, and from top to bottom in each column, as shown in Figure 1. *State* has four columns, which we refer to as *State*[0], *State*[1], *State*[2], and *State*[3], from left to right.

3.1.3 Counter Mode

In the case of counter mode, there is no need to implement a decryption algorithm of the block cipher. As shown in Figure 2, a 16 byte data block, called a counter, is encrypted by a block cipher in the place of plaintext. Then, this result and plaintext are merged by an XOR operation to create ciphertext. An n-bit counter is typically initialized to a pre-defined value (`IV`), and is then increased based on a pre-defined rule. The sequence of counter values must be distinguished from each other. In other words, the counter values must all have different values.

Table 1: Notation

Notation	Description
$X_{i,j}$	input (*rin*) or output (*rout*) *State* (i^{th} block, j^{th} round) (*rin* / *rout* is represented by X)
in_i	i^{th} byte of input data (counter value in CTR mode) block
$rk_{i,j}$	byte of round key (key for i^{th} round, j^{th} byte)
$S_{i,j,k}$	byte of *State* (i^{th} block, j^{th} round, k^{th} byte)
$S[k]$	k^{th} byte of input *State*
$X_{i,j}[0]$	1^{st} column of $X_{i,j}$
$X_{i,j}[1]$	2^{nd} column of $X_{i,j}$
$X_{i,j}[2]$	3^{rd} column of $X_{i,j}$
$X_{i,j}[3]$	4^{th} column of $X_{i,j}$

3.1.4 Bitslice Implementation

The bitslice implementation technique was initially proposed by Biham to improve the software performance of DES [Bih97]. The bitslice technique simulates a hardware implementation in software, and all operations are expressed as a sequence of Boolean operations. On x86 processors, this technique is not practical for improving the performance of AES. However, on an x64 architecture, it is considered an interesting topic. A method to implement bitslice AES on x64 platforms was first reported in [Mat06]. Until then, 128-bit XMM registers were of no use, owing to their poor performance caused by treating a 128-bit instruction as two 64-bit operations on the processors (Pentium 4 and Athlon64). However, with Intel Core 2 processors, bitsliced AES is implemented to fully utilize XMM instructions [MN07], [KS09].

3.2 Notation

We use the notation described in Table 1 throughout the paper. We denote $X_{i,j}$ as the input or output *State* of the j^{th} round of the i^{th} block. For example, $rin_{3,1}$ denotes the input *State* of the first round in the fourth block. This round function produces $rout_{3,1}$. The i^{th} byte of the input data block is denoted by in_i and $rk_{i,j}$ refers to the j^{th} byte of the round key corresponding to the i^{th} round. A single byte of *State* is represented by $S[k]$. Because the block size of AES is 16 bytes, S[0] indicates the most significant byte (MSB) and S[15] indicates the least significant byte (LSB) of *State*. As mentioned above, $X_{i,j}[i]$ denotes the i^{th} column of the input or output *State* from left to right. According to the AES block-to-state transformation, $X_{i,j}[i]$ is comprised of S$[i*4]$, S$[i*4+1]$, S$[i*4+2]$, and S$[i*4+3]$, where $i = 0, 1, 2, 3$. We define $S_{i,j,k}$ as the single byte of the particular *State* that indicates the k^{th} byte of the j^{th} round of the i^{th} block.

4 Implementation Technique Using Repetitive Data: FACE

In this section, we present our implementation techniques to enhance the efficiency of AES CTR mode encryption. We reuse the repetitive partial data contained in the output *State* or intermediate calculation results of the round function, from round 0 (i.e. initial whitening. We denoted the initial whitening as 'round 0' for generalizing the naming rules of targets) to round 2. In this paper, we present five types of reuse techniques, which are briefly described below.

- FACE$_{rd0}$: This technique caches the result-*State* of an initial whitening (round 0) and reuses the cached data at the next block. While the overlapping area of the

result-*State* is 15 bytes, we cache and reuse only 12 bytes of the result-*State* to minimize the update frequency of the cache. Section 4.1 explains this technique and its benefit.

- FACE_{rd1}: This phase was already introduced in [BS08, Lib, Wu07]. We denote this technique as FACE_{rd1} and describe it in Section 4.2.

- FACE_{rd1+}: In AES round 1, FACE_{rd1} caches and reuses 12 bytes of the result-*State*. Originally, the remaining 4 bytes should be recalculated in every block. FACE_{rd1+} is a technique that generates pre-computation values for these 4 bytes of the result-*State*, which are not covered by FACE_{rd1}. These values can be generated either before or during encryption. Section 4.3 explains this technique.

- FACE_{rd2}: This technique caches and reuses 16 bytes of *State* after the `MixColumns()` and `AddRoundKey()` transformations in round 2. Our implementation saves 16 instructions (8 integer + 8 load) more as compared with [BS08, Lib, Wu07]. Section 4.4 gives a detailed explanation of this technique.

- FACE_{rd2+}: In AES round 2, FACE_{rd2} caches and reuses 16 bytes of the intermediate calculation result. Similar to round 1, the remaining part should be recalculated in every block. Then, this calculated remaining part and the cached data of FACE_{rd2} are merged by XOR operations to complete round 2. FACE_{rd2+} generates pre-computation values for this remaining calculation result, which is not covered by FACE_{rd2}. These values also can be generated either before or during encryption. Section 4.5 accounts for this technique.

Existing counter-mode caching method only just deals with partial result of each round transformation. It is suitable only for table-based implementation to show its efficiency because the remaining result should be recalculated in every block. However, FACE can cover a round transformation entirely. FACE can be applied efficiently to existing implementation methods (table-based and bitsliced) and even to AES-NI.

We elucidate our methods using the general AES model in order to represent each method convincingly.

4.1 Technique Applied to Initial Whitening (FACE_{rd0})

The input value of the first block is initialized to a pre-defined value (i.e., `IV`) in the AES CTR mode. While processing multiple blocks of plaintext, the value of `IV` increases, but changes only the last byte of `IV` in most cases. This means that the only difference between a given block and the previous block is the last byte, unless the last byte exceeds `0xFF`. For example, if $\texttt{IV}_0$ (the input value of the first block) is initialized to {`0x00000000`, `0x00000000`, `0x00000000`, `0x00000001`}, $\texttt{IV}_0$ and $\texttt{IV}_1$ (the input value of the next block, which is increased by the counter) have the same value, except in last 1 byte, as in the example below.

· $\texttt{IV}_0$: `0x00000000 0x00000000 0x00000000 0x00000001`
· $\texttt{IV}_1$: `0x00000000 0x00000000 0x00000000 0x00000002`

In this case, most bytes do not change until the value of the last byte reaches `0xFF`. We use this property to reuse the result of the operation in initial whitening (round 0). The process of the initial whitening is as follows:

· Input value = { $in_0, in_1, \ldots, in_{14}, in_{15}$ }
· Round key$_0$ = { $rk_{0,0}, rk_{0,1}, \ldots, rk_{0,14}, rk_{0,15}$ }
· Output *State* of initial whitening = { $S[0], \ldots, S[15]$ } ($S[i] = in_i \oplus rk_{0,i}$, for $0 \le i \le 15$)

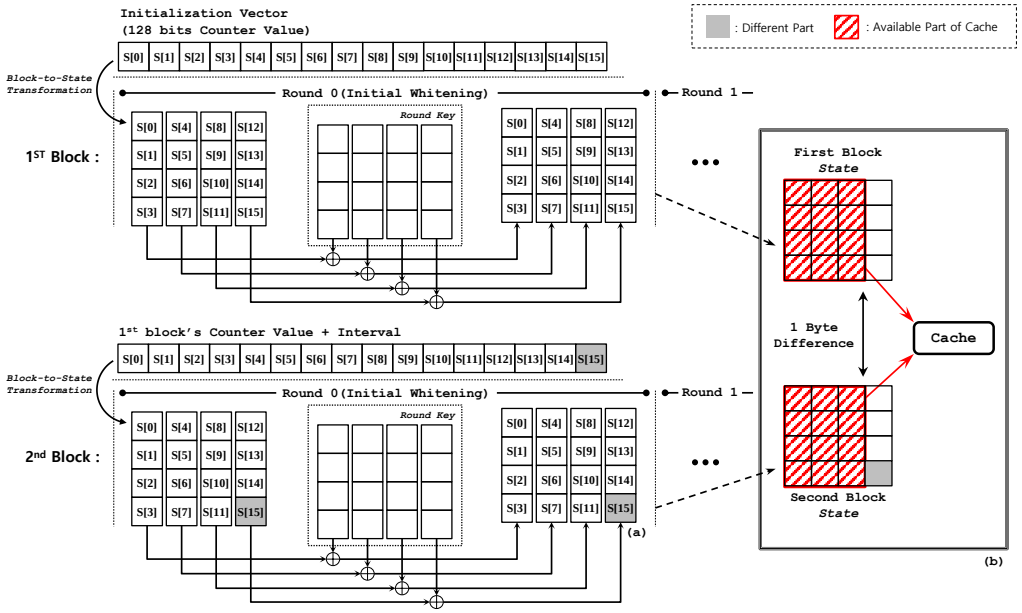

Figure 3: FACE$_{rd0}$: AES initial whitening (round 0) of the first block and the second block in CTR mode. (a) shows the 1 byte difference between $rout_{i,0}$ and $rout_{i+1,0}$. (b) indicates the cached part used by FACE$_{rd0}$.

In keeping with the above procedure, the following presents the process for the first block and the second block in the initial whitening (round 0).

· First block input ($\texttt{IV}_0$) = $\{in_0, in_1, \ldots, in_{14}, in_{15}\}$
· Second block input ($\texttt{IV}_1$) = $\{in_0, in_1, \ldots, in_{14}, in'_{15}\}$
· Round key$_0$ = $\{rk_{0,0}, rk_{0,1}, rk_{0,2}, \ldots, rk_{0,14}, rk_{0,15}\}$
· Initial whitening output *State* of first block ($\texttt{IV}_0 \oplus$ Round key$_0$) = $\{S[0], S[1], \ldots, S[14], S[15]\}$
· Initial whitening output *State* of second block ($\texttt{IV}_1 \oplus$ Round key$_0$) = $\{S[0], S[1], \ldots, S[14], S'[15]\}$

From results such as $\texttt{IV}_0$ and $\texttt{IV}_1$, we find that the first block output and second block output differ in the last 1 byte ($S[15]$ and $S'[15]$). FACE$_{rd0}$ reuses only $rout_{i,0}[0]$, $rout_{i,0}[1]$, and $rout_{i,0}[2]$ to minimize cache updating. Therefore, while operating 2^{32}-1 successive blocks, ours does not need to update the cache information. Figure 3 indicates the process for the initial whitening (round 0) and the part of *State* cached from the round operation result. FACE$_{rd0}$ can use the same cached information until the last four bytes of `IV` reach `0xFFFFFFFF` after which the cache of FACE$_{rd0}$ should be updated. For example, if the counter is increased by 1 and the `IV` value is 0, the block that causes a carry at the 11^{th} byte of the counter value is the 4,294,967,297th block. In this case, the cached information can be used while calculating 4,294,967,295 blocks instead of performing XOR operations. That is, FACE$_{rd0}$ updates the cached data only once throughout the 65.5 GB of plaintext.

4.2 Technique Applied to Round 1 (FACE$_{rd1}$)

The output *State* of the initial whitening (round 0) is used as the input of round 1. In round 1, FACE$_{rd1}$ can also cache and reuse data. Figure 4 depicts FACE$_{rd1}$. The feature of FACE$_{rd1}$ is that $rin_{0,1}$ and $rin_{1,1}$ are different only in the last 1 byte, as shown below.

- $rout_{i,0} = rin_{i,1}$
- Comparison between $rin_{0,1}$ and $rin_{1,1}$
  · $S_{0,1,k} = S_{1,1,k}$, for $0 \leq k \leq 14$

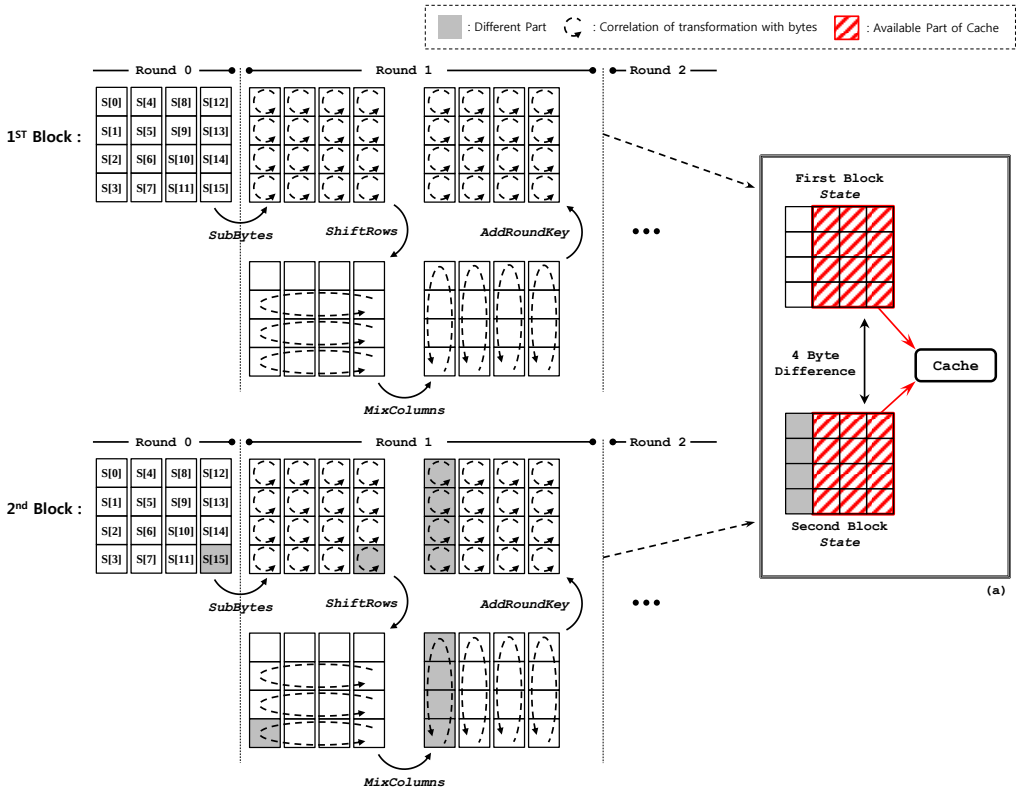

Figure 4: FACE_{rd1}: The process of AES round 1 and the diffusion of the difference between the first block and the second block. (a) represents the cached part used by FACE_{rd1}.

· $S_{0,1,k} \neq S_{1,1,k}$, if $k = 15$

In round 1, the difference in the last 1 byte of the input *State* affects only $rout_{i,1}[0]$ instead of affecting the whole *State*. Therefore, as shown in Figure 4, FACE_{rd1} can cache and reuse the result of round 1 except for $rout_{i,1}[0]$. The following presents the process of the first block and the second block in round 1.

· $rin_{0,1} = \{ S_{0,1,0} , S_{0,1,1} , \ldots , S_{0,1,14} , S_{0,1,15} \}$
· $rin_{1,1} = \{ S_{0,1,0} , S_{0,1,1} , \ldots , S_{0,1,14} , S_{1,1,15} \}$

are changed by four transformations to

· $rout_{0,1} = \{ S'_{0,1,0} , S'_{0,1,5} , S'_{0,1,10} , S'_{0,1,15} ,$
$S'_{0,1,4} , S'_{0,1,9} , S'_{0,1,14} , S'_{0,1,3} ,$
$S'_{0,1,8} , S'_{0,1,13} , S'_{0,1,2} , S'_{0,1,7} ,$
$S'_{0,1,12} , S'_{0,1,1} , S'_{0,1,6} , S'_{0,1,11} \}$

· $rout_{1,1} = \{ S'_{1,1,0} , S'_{1,1,5} , S'_{1,1,10} , S'_{1,1,15} ,$
$S'_{0,1,4} , S'_{0,1,9} , S'_{0,1,14} , S'_{0,1,3} ,$
$S'_{0,1,8} , S'_{0,1,13} , S'_{0,1,2} , S'_{0,1,7} ,$
$S'_{0,1,12} , S'_{0,1,1} , S'_{0,1,6} , S'_{0,1,11} \}$

- Comparison between $rout_{0,1}$ and $rout_{1,1}$
· $rout_{0,1}[\mathrm{i}] \neq rout_{1,1}[\mathrm{i}]$, if $i = 0$
· $rout_{0,1}[\mathrm{i}] = rout_{1,1}[\mathrm{i}]$, for $1 \leq i \leq 3$

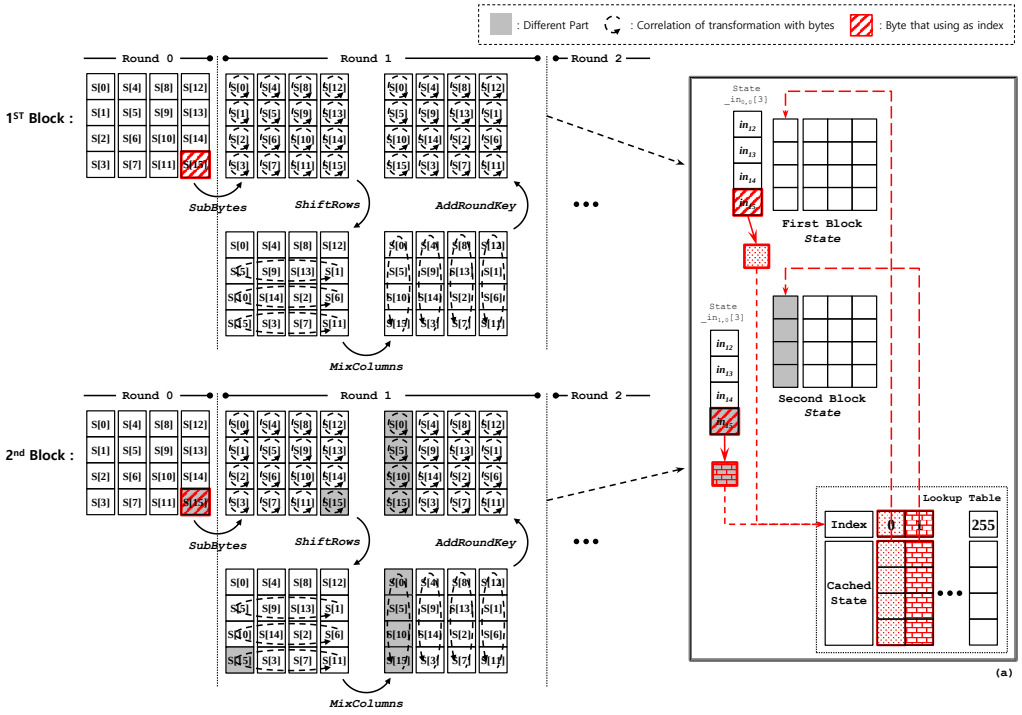

Figure 5: FACE$_{rd1+}$: The process of AES round 1 and FACE$_{rd1+}$. (a) shows how to refer the pre-computation values.

Thus, we can cache $rout_{0,1}$[i] ($1 \leq i \leq 3$) and reuse them for the next block. Because $rout_{i,1}[1]$ is changed when $S[14]$ of $rin_{i,1}$ is altered, FACE$_{rd1}$ updates the cached data once for every 256 (2^8) blocks.

4.3 Additional Technique Applied to Round 1 (FACE$_{rd1+}$)

In general, the implemented cryptographic algorithm in a real environment comprises two phases. The first phase is an initialization stage. In this stage, the cryptographic algorithms accept cryptographic parameters such as a secret-key and IV. In the case of AES, the secret-key is expanded to the round key as the AES key length, and `IV` is used as a counter in CTR mode. The second phase is the encryption/decryption stage. Input data are encrypted or decrypted using a fixed round key and `IV`. Once the cryptographic parameters are initialized, phase 2 can perform encryption or decryption processes using the same parameters. In many real environments, phase 1 is called only once, but phase 2 is performed several times.

In this section, we propose a technique that generates pre-computation values for $rout_{i,1}[0]$ in phase 1 that are reused for round 1 in phase 2; if the length of the input data exceeds 256 blocks, pre-computation values can be generated and reused in phase 2. As mentioned in Section 4.2, most of the result-*State* of round 1 ($rout_{i,1}[1]$, $rout_{i,1}[2]$, and $rout_{i,1}[3]$) are covered by FACE$_{rd1}$. However, $rout_{i,1}[0]$ is changed for every block because the alteration of in_{15} affects all bytes of $rout_{i,1}[0]$. Figure 7 shows the process of the *State* transformation in round 1 while the AES CTR mode operates. As shown in Figure 7, the factors that determine $rout_{i,1}[0]$ are $S[0]$, $S[5]$, $S[10]$, and $S[15]$ of $rin_{i,1}$. As already shown, S[15] of $rin_{i,1}$ is $in_{15} \oplus rk_{0,15}$. Because the round key remains the same while working, S[15] of $rin_{i,1}$ is determined by in_{15}. According to the increasing rule that adds

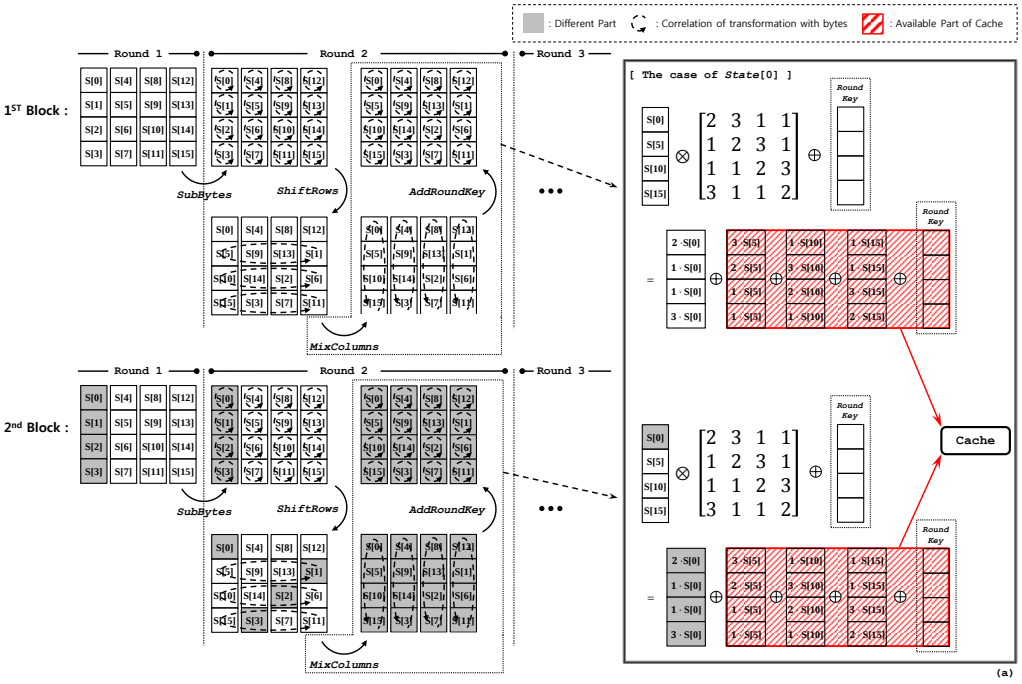

Figure 6: FACE_{rd2}: The diffusion process of the difference between the first block and the second block in round 2. (a) indicates the `MixColumns()` and `AddRoundKey()` transformations using *State*[0]. It also shows the available part of the cache.

one to the previous counter value, in_{15} is changed continuously. However, in_{10} is altered only when the value of in_{11}-in_{15} exceeds `0xFFFFFFFFFF`. This means that in_0, in_5, and in_{10} are never changed while processing 1,099,511,627,776 blocks (16 TB). Thus, we can generate a temporary look-up table that uses in_{15} as an index. If we create a table that determines $rout_{i,1}[0]$, we need an additional 1 KB (4×256) of memory because $rout_{i,1}[0]$ is 4 bytes and in_{15} changes from `0x00` to `0xFF`. Although this table is not the same for every crypto instance, it can be pre-computed and determined in the initialization stage because it depends on the secret-key and `IV`. The initialization stage is performed once every instance, so we can improve the efficiency in the encryption/decryption stage using FACE_{rd1+}.

Pre-computation values can be used until the change of in_{15} influences in_{10}, which occurs when in_{10} is changed on the next block by [in_{11}-in_{15}] becoming [`0xFFFFFFFFFF`]. At this time, the pre-computation values must be updated. Updated values can be used for processing 2^{40} blocks without the need for an additional updating process. Thus, the efficiency of the calculation is improved.

4.4 Technique Applied to Round 2 (FACE_{rd2})

In round 2, the result-*State* of round 1 is used as the input of round 2. Now, the difference in the number of bytes between $rin_{0,2}$ and $rin_{1,2}$ is four.

- $rin_{0,2}[\mathrm{i}] \neq rin_{1,2}[\mathrm{i}]$, if $i = 0$
- $rin_{0,2}[\mathrm{i}] = rin_{1,2}[\mathrm{i}]$, for $1 \leq i \leq 3$

Figure 6 describes the round 2 operation for the first and second blocks. At the end of round 2, the difference in *State* between $rin_{0,2}$ and $rin_{1,2}$ affects the whole byte of the

result-*State* of round 2. Therefore, $rout_{0,2}$ and $rout_{1,2}$ are completely different. However, as in the other preceding techniques, we can also cache intermediate information in round 2.

The following expresses the `MixColumns()` and `AddRoundKey()` transformations using *State*[0] as an example. Before the `MixColumns()` transformation, *State*[0] is comprised of {$S[0]$, $S[5]$, $S[10]$, and $S[15]$} due to the `ShiftRows()` transformation.

$$\begin{pmatrix} S'[0] \\ S'[5] \\ S'[10] \\ S'[15] \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \otimes \begin{pmatrix} S[0] \\ S[5] \\ S[10] \\ S[15] \end{pmatrix} \oplus \begin{pmatrix} rk_{2,0} \\ rk_{2,1} \\ rk_{2,2} \\ rk_{2,3} \end{pmatrix}$$
$$= \begin{pmatrix} 2 \cdot S[0] \oplus 3 \cdot S[5] \oplus 1 \cdot S[10] \oplus 1 \cdot S[15] \oplus rk_{2,0} \\ 1 \cdot S[0] \oplus 2 \cdot S[5] \oplus 3 \cdot S[10] \oplus 1 \cdot S[15] \oplus rk_{2,1} \\ 1 \cdot S[0] \oplus 1 \cdot S[5] \oplus 2 \cdot S[10] \oplus 3 \cdot S[15] \oplus rk_{2,2} \\ 3 \cdot S[0] \oplus 1 \cdot S[5] \oplus 1 \cdot S[10] \oplus 2 \cdot S[15] \oplus rk_{2,3} \end{pmatrix}$$

Upon processing the `MixColumns()` transformation, as shown in Figure 6(a), the data related to $S[0]$ would change in $rout_{i,2}[0]$, but the other data do not. Therefore, we can cache the intermediate calculation result for $rout_{i,2}[0]$ as follows:

- Cached part of $rout_{i,2}[0]$: $\begin{pmatrix} 3 \cdot S[5] \oplus 1 \cdot S[10] \oplus 1 \cdot S[15] \oplus rk_{2,0} \\ 2 \cdot S[5] \oplus 3 \cdot S[10] \oplus 1 \cdot S[15] \oplus rk_{2,1} \\ 1 \cdot S[5] \oplus 2 \cdot S[10] \oplus 3 \cdot S[15] \oplus rk_{2,2} \\ 1 \cdot S[5] \oplus 1 \cdot S[10] \oplus 2 \cdot S[15] \oplus rk_{2,3} \end{pmatrix}$

This technique can be expanded to *State*[1], *State*[2] and *State*[3]. That is, State[1], State[2], and State[3] have changed parts that are related to $S[3]$, $S[2]$, and $S[1]$ respectively. Thus FACE_{rd2} caches the rest of the intermediate information, as follows.

- In the case of *State*[1], cached part of $rout_{i,2}[1]$: $\begin{pmatrix} 2 \cdot S[4] \oplus 3 \cdot S[9] \oplus 1 \cdot S[14] \oplus rk_{2,4} \\ 1 \cdot S[4] \oplus 2 \cdot S[9] \oplus 3 \cdot S[14] \oplus rk_{2,5} \\ 1 \cdot S[4] \oplus 1 \cdot S[9] \oplus 2 \cdot S[14] \oplus rk_{2,6} \\ 3 \cdot S[4] \oplus 1 \cdot S[9] \oplus 1 \cdot S[14] \oplus rk_{2,7} \end{pmatrix}$

- In the case of *State*[2], cached part of $rout_{i,2}[2]$: $\begin{pmatrix} 2 \cdot S[8] \oplus 3 \cdot S[13] \oplus 1 \cdot S[7] \oplus rk_{2,8} \\ 1 \cdot S[8] \oplus 2 \cdot S[13] \oplus 1 \cdot S[7] \oplus rk_{2,9} \\ 1 \cdot S[8] \oplus 1 \cdot S[13] \oplus 3 \cdot S[7] \oplus rk_{2,10} \\ 3 \cdot S[8] \oplus 1 \cdot S[13] \oplus 2 \cdot S[7] \oplus rk_{2,11} \end{pmatrix}$

- In the case of *State*[3], cached part of $rout_{i,2}[3]$: $\begin{pmatrix} 2 \cdot S[12] \oplus 1 \cdot S[6] \oplus 1 \cdot S[11] \oplus rk_{2,12} \\ 1 \cdot S[12] \oplus 3 \cdot S[6] \oplus 1 \cdot S[11] \oplus rk_{2,13} \\ 1 \cdot S[12] \oplus 2 \cdot S[6] \oplus 3 \cdot S[11] \oplus rk_{2,14} \\ 3 \cdot S[12] \oplus 1 \cdot S[6] \oplus 2 \cdot S[11] \oplus rk_{2,15} \end{pmatrix}$

FACE_{rd2} can reuse these cached data while processing 255(2^8-1) consecutive blocks. The frequency of updates is equal to that of FACE_{rd1} because the alteration of $rin_{i,2}[1]$ affects the cached data.

4.5 Additional Technique Applied to Round 2 (FACE_{rd2+})

In this section, we propose a technique that generates pre-computation values for the remaining intermediate data, which is not covered by FACE_{rd2}. As we mentioned in section 4.3, the pre-computation values for FACE_{rd2+} can also be generated either before or during encryption (in phase 1 or phase 2, respectively).

As described in section 4.4, $rout_{i,2}[0]$ is calculated after the `SubBytes()` and `ShiftRows()` transformations as :

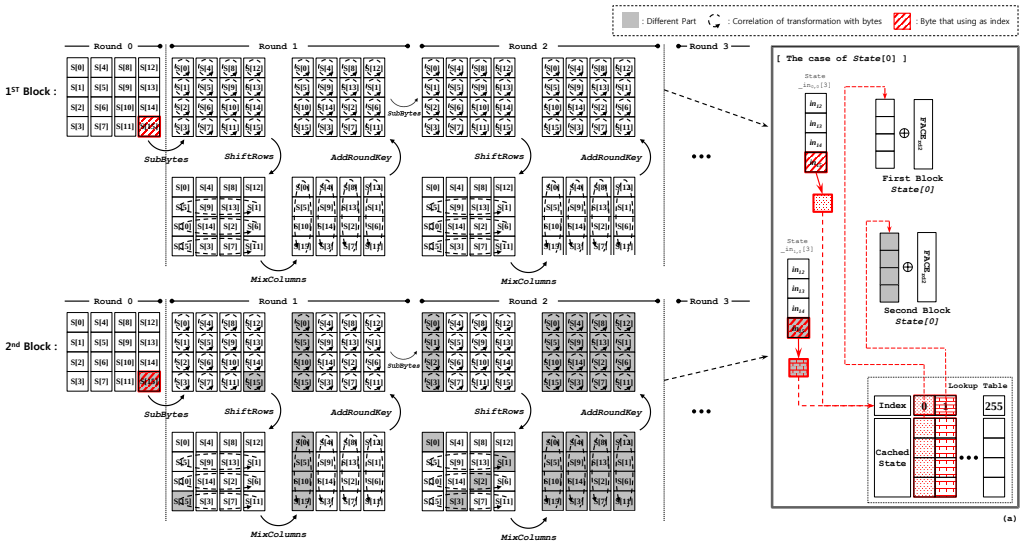

Figure 7: FACE$_{rd2+}$: The process of AES round 1, round 2, and FACE$_{rd2+}$. (a) shows how to refer the pre-computation values and how to calculate the result of round 2.

$$\begin{pmatrix} S'[0] \\ S'[5] \\ S'[10] \\ S'[15] \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \otimes \begin{pmatrix} S[0] \\ S[5] \\ S[10] \\ S[15] \end{pmatrix} \oplus \begin{pmatrix} rk_{2,0} \\ rk_{2,1} \\ rk_{2,2} \\ rk_{2,3} \end{pmatrix}$$
$$= \begin{pmatrix} 2 \cdot S[0] \oplus 3 \cdot S[5] \oplus 1 \cdot S[10] \oplus 1 \cdot S[15] \oplus rk_{2,0} \\ 1 \cdot S[0] \oplus 2 \cdot S[5] \oplus 3 \cdot S[10] \oplus 1 \cdot S[15] \oplus rk_{2,1} \\ 1 \cdot S[0] \oplus 1 \cdot S[5] \oplus 2 \cdot S[10] \oplus 3 \cdot S[15] \oplus rk_{2,2} \\ 3 \cdot S[0] \oplus 1 \cdot S[5] \oplus 1 \cdot S[10] \oplus 2 \cdot S[15] \oplus rk_{2,3} \end{pmatrix}$$

Except for the intermediate result that is cached by FACE$_{rd2}$, the remaining data are $2 \cdot S[0]$ for $S'[0]$, $1 \cdot S[0]$ for $S'[5]$, $1 \cdot S[0]$ for $S'[10]$, and $3 \cdot S[0]$ for $S'[15]$. All of this remaining data are related to S[0]. Similar situations can be seen in other $rout_{i,2}[i]$ $(1 \leq i \leq 3)$. Similarly, the remaining data of $rout_{i,2}[1]$, $rout_{i,2}[2]$, and $rout_{i,2}[3]$ are related to $S[3]$, $S[2]$, and $S[1]$ respectively. Recall that the `SubBytes()` and `ShiftRows()` transformations do not affect other bytes. Thus, these four bytes ($S[0]$, $S[1]$, $S[2]$, $S[3]$) are considered to be the same as $rin_{i,2}[0]$. This means that the remaining intermediate data is determined by $rin_{i,2}[0]$.

According to FACE$_{rd1+}$, $rout_{i,1}[0]$ ($rin_{i,2}[0]$) is determined by in_{15}. And the remaining intermediate data for FACE$_{rd2+}$ is determined by $rin_{i,2}[0]$. In the end, we can generate a temporary look-up table and use in_{15} as an index. If we create a pre-computation table for FACE$_{rd2+}$, we need an additional 4 KB (16×256) of memory because the remaining intermediate data is 4 bytes for each $rout_{i,2}[i]$ $(0 \leq i \leq 3)$, and in_{15} changes from `0x00` to `0xFF`. FACE$_{rd2+}$ can reuse these cached data while processing 2^{40} consecutive blocks. The frequency of updates is equal to that of FACE$_{rd1+}$.

5 Evaluations

5.1 Implementation

The proposed method (FACE) can be employed in any AES CTR implementation, regardless of the platform, environment, or implementation method. We implement FACE by

Table 2: Environments used for evaluation

	Test Environment 1	Test Environment 2	Test Environment 3
CPU	Intel Core 2 Quad Q9550	Intel Core i7 4770K	Intel Core i7 8700K
CPU Frequency	2.8 GHz	3.5 GHz	3.7 GHz
RAM	4 GB	8 GB	16 GB
OS	Linux 3.19.0-32 x86_64	Linux 3.19.0-32 x86_64	Linux 4.13.0-36 x86_64

modifying the AES source code contained in the open-source libraries (OpenSSL [Pro] and Crypto++ [Lib]). OpenSSL provides AES source code for both the table-based and the bitsliced cases. In particular, bitsliced AES is implemented based on [KS09], which is known as the fastest bitsliced AES CTR software implementation with one core. We analyzed the source code of [KS09] and compared it with the bitsliced implementation of OpenSSL. As a result, we found that the two codes are very similar and almost identical (as OpenSSL commented on its source code). Moreover, OpenSSL left a record of its own comparison with [KS09], and OpenSSL shows better performance. We think OpenSSL is more practical because it supports 192- and 256-bit keys, unlike [KS09]. Thus, we selected OpenSSL as our comparison target of bitsliced implementation. The AES implementation of Crypto++ supports AES-NI instructions via compiler intrinsics. It provides two encryption interfaces; one is for processing one 128-bit block per call (which we denoted as "1 x 1"), and the other is for processing four 128-bit blocks per call (which we denoted as "4 x 1"). To the best of our knowledge (based on eSTREAM/Crypto++ benchmark and other literature), AES-NI-based implementation of Crypto++ can be considered the fastest one. Thus, we selected Crypto++ as our comparison target of AES-NI-based implementation. For the experiments on AES-NI-based FACE, we prepared two kinds of implementations: R1 and R2. R1 is an implementation, which leverages FACE up to round 1. R2 is an implementation, which adopts FACE up to round 2.

For a fair comparison, we did not re-code the existing strategy into our own implementation. Since the same strategy can record different performance depending on the quality of code, and consequently, it can be misunderstood that the reason of improvements comes from the quality of the code we produced. Thus, we opted to adopt an existing implementation and use it as our own. And then we made only minor modifications to the existing code in order to apply our strategy. Except for our new strategy, all other conditions remain the same. This can be confirmed by comparing our target open-source with the appendix.

The five techniques of FACE can be chosen selectively, with consideration for the environment in which FACE will be applied. When code size or cache space is an issue, our techniques can be applied by choosing the most efficient combinations.

As a result of applying FACE to existing implementations, table-based FACE needs 12 instructions up to round 2, whereas the existing implementation [Pro] needs 128 instructions. Considering the cache update process, which occurs once for every 256 blocks, table-based FACE needs 12.4 instructions on average. Bitsliced FACE requires 74 instructions up to round 2, whereas [KS09] requires 618 instructions. As with the table-based FACE, considering the cache update procedure, bitsliced FACE needs 91.9 instructions on average. AES-NI-based FACE requires 1 intrinsic instruction, 1 memory reference, and 1 arithmetic instruction (simply, 3 instructions) up to round 2, whereas [Lib] requires 3 intrinsic instructions and 3 memory references (simply, 6 instructions). According to the Intel instruction latency and throughput [Cor18], a briefly calculated throughput (required cycle) for AES-NI-based FACE is 0.83 up to round 2, whereas for [Lib] it is 3.08. Considering the cache update procedure as it is in other methods, AES-NI-based FACE requires (simply) 3.02 instructions on average.

Table 3: Performance comparison of AES-CTR implementations in cycles/byte

Platform	Implementation Method		Target	Input (Plaintext) Size											
				1024 bytes			4096 bytes			20480 bytes			40960 bytes		
				128	192	256	128	192	256	128	192	256	128	192	256
Test Env 1	Table-based		OpenSSL	15.849	18.302	20.710	15.786	18.272	20.680	15.766	18.249	20.665	15.768	18.238	20.659
			This Paper	12.452	14.947	17.336	12.407	14.936	17.329	12.394	14.911	17.321	12.399	14.913	17.326
	Bitsliced		[KS09]	8.014	9.495	10.960	7.811(7.59)	9.251	10.686	7.763	9.195	10.624	7.764	9.192	10.618
			This Paper	6.754	8.180	9.607	**6.408(6.347)**	7.797	9.180	6.364	7.755	9.119	6.360	7.752	9.108
Test Env 2	Table-based		OpenSSL	10.562	12.309	14.036	10.553	12.348	14.067	10.529	12.276	14.023	10.528	12.276	14.023
			This Paper	8.380	10.085	11.808	8.344	10.064	11.797	8.371	10.067	11.810	8.368	10.071	11.808
	Bitsliced		[KS09]	5.687	6.745	7.803	5.530	6.554	7.573	5.514	6.491	7.511	5.500	6.482	7.495
			This Paper	4.696	5.737	6.787	4.429	5.455	6.476	4.398	5.407	6.425	4.397	5.406	6.422
	AES-NI	1 x 1	Crypto++	2.540	2.957	3.321	2.506	2.896	3.283	2.698	3.083	3.482	2.695	3.080	3.477
			This Paper (R1)	1.025	1.267	1.556	1.018	1.253	1.552	1.073	1.301	1.578	1.071	1.294	1.558
			This Paper (R2)	0.927	1.160	1.383	0.917	1.146	1.377	1.040	1.188	1.398	1.040	1.189	1.398
		4 x 1	Crypto++	0.730	0.861	0.984	0.704	0.840	0.983	0.688	0.824	0.969	0.684	0.822	0.967
			This Paper (R1)	0.634	0.781	0.923	0.623	0.769	0.920	0.621	0.765	0.911	0.620	0.765	0.910
			This Paper (R2)	0.592	0.727	0.869	0.580	0.714	0.858	0.578	0.711	0.857	0.578	0.711	0.857
Test Env 3	Table-based		OpenSSL	9.374	10.948	12.645	9.223	10.788	12.496	9.083	10.354	11.822	8.716	10.087	11.644
			This Paper	7.185	8.741	10.346	7.114	8.726	10.230	7.081	8.408	9.847	6.855	8.203	9.647
	Bitsliced		[KS09]	5.273	6.108	7.254	5.172	6.074	7.079	5.097	5.999	6.995	5.032	5.879	6.952
			This Paper	4.339	5.356	6.278	3.932	4.984	5.987	4.006	4.945	5.873	3.812	4.691	5.571
	AES-NI	1 x 1	Crypto++	1.665	1.871	2.059	1.625	1.847	2.043	1.617	1.832	2.029	1.611	1.807	2.021
			This Paper (R1)	0.778	0.867	0.986	0.739	0.827	0.959	0.737	0.822	0.956	**0.726**	0.819	0.948
			This Paper (R2)	0.703	0.786	0.880	0.662	0.775	0.867	0.659	0.732	0.874	**0.658**	0.733	0.876
		4 x 1	Crypto++	0.551	0.669	0.767	0.547	0.642	0.758	0.537	0.636	0.745	0.531	0.622	0.739
			This Paper (R1)	0.513	0.607	0.706	0.494	0.586	0.698	0.483	0.581	0.684	**0.473**	0.573	0.677
			This Paper (R2)	0.450	0.547	0.638	0.441	0.533	0.636	0.442	0.539	0.624	**0.434**	0.539	0.625

We note that the code used in the experiments is identical, regardless of the test environment. For example, we do not use additional optimization techniques for the bitsliced FACE in the test environment 2 and 3 (microarchitectures that provide a 256-bit AVX instruction set). This is guaranteed because the bitsliced method is implemented using low-level programming language (Assembly). In conclusion, when implementing FACE, we do not take advantage of other optimizations that leverage special features of the test environment.

5.2 Experimental Results

To evaluate our method, FACE, we first verify the proposed technique and its implementation using test vectors with different input lengths and key lengths. The results show that the table-based, bitsliced, and AES-NI-based FACE all work correctly. Then, we measure the throughput of FACE that was employed in table-based, bitsliced, and AES-NI-based implementations on Intel Core 2 Quad and Core i7 processors. Then, we compare our results to those of a default implementation contained in the open-source libraries. A description of the environments used for the evaluation is given in Table 2. All tests were conducted using only one core.

A comprehensive comparison of the evaluation results is summarized in Table 3. We measure the throughput using three key lengths (128, 192, and 256 bits), while changing the size of the input blocks to 1024, 4096, 20480, and 40960 bytes. We also measure the performance of the base code that FACE leveraged, within the same environments. Since all the experimental environments can not be the same, all experimental results, except those recorded in the literature using the same environment, are presented with our own measuring results in our environments.

In the case of the table-based implementation, our implementation of AES CTR

achieves a performance improvement of $15\% - 20\%$. This result shows that table-based FACE achieves similar performance to AES CTR, which has lower-level security (e.g., the performance of table-based FACE-192 $\approx$ the performance of the default AES-128). We compared our results with table-based implementation of OpenSSL. However, there exists an outperformed result, which records 10.57 cycles/byte on Intel Core 2 Quad Q9550 [BS08]. We did not try to apply our strategy to [BS08]. Because our main targets are bitsliced and AES-NI-based implementations. Additionally, we believe that the table-based implementation is still not of much interest; it has reached its performance limits, and also has a security concern. We intended to simply show, using the experiments, that our proposal can be applied to table-based. It was not our goal to achieve state-of-the-art results with the table-based implementation.

Bitsliced FACE needs 6.41 cycles/byte for an input size of 4096 bytes and a 128 bit key length, where the previous work recorded 7.81 cycles/byte in test environment 1. It appears that the performance of [KS09] (7.81 cpb) is worse than the reported throughput in [KS09] (7.59 cpb). Our insight into this is that the results of our experiments include key transformation (a conversion of round keys from a table-based representation to bitslice form) at the beginning of the encryption phase (the result of [KS09] did not include such key transformation cost in the encryption phase). Also, for a common usage, [KS09] implementation of OpenSSL adds more routines to support other key lengths (192 and 256 bits) and employs different byte order. If we exclude the key transformation process, [KS09] implementation of OpenSSL requires 7.75 cycles/byte according to our experiments. Bitsliced FACE without key transformation records 6.35 cycles/byte). Further, bitsliced FACE needs 3.93 cycles/byte, while [KS09] implementation of OpenSSL records 5.17 cycles/byte on a recent Intel Core i7 (Test Environment 3). Our bitsliced implementation of AES CTR is about 20% faster than those in previous works, and the result of 6.41 cycles/byte for an input size of 4096 bytes is the highest throughput ever achieved in a PC environment (without AVX or AES-NI). Thanks to the characteristic of FACE that uses repetitive data, the throughput of FACE increases as the length of the input block increases. Finally, bitsliced FACE records 6.36 cycles/byte for an input size of 40960 bytes with a 128-bit key.

Our AES-NI-based FACE needs 0.44 cycles/byte for an input size of 4096 bytes and a 128 bit key length, where AES-NI-based AES CTR implementation of [Lib] recorded 0.54 cycles/byte on Intel Core i7 8700K. Further, AES-NI-based FACE records 0.43 cycles/byte for an input size of 40960 bytes with a 128-bit key, whereas [Lib] needs 0.53 cycles/byte. The result of 0.44 cycles/byte is also the highest throughput ever achieved in an AES CTR implementation with AES-NI.

The speedups shown in our experiments are on the same level as (or even exceed) the simple theoretical model of simply skipping the first two rounds. FACE targets the first two rounds of AES. AES-128 has 10 rounds and AES-256 has 14 rounds. If the first two rounds could be skipped completely, the expectable speedups are up to 20% and 14%, respectively. Our insight into the reason of speedups, which are shown to be as good as this kind of simplistic model, can be described briefly as follows. First, FACE can be considered as completely skipping the first two rounds like the simple theoretical model. After only 1 XOR operation, the first two rounds are completed, and this cost is equal to initial whitening. Second, FACE additionally skips several operations; copying input to local variable, increasing counter value (if the increment is pre-defined, there is no need to increase counter for every block), and in case of bitslice method, a costly transformation of input to bitslice form as well as ShiftRows of round 3. Such operations are also included in the cached value of FACE. Third, FACE has benefits (only for table-based) from the optimization of compiler due to its simplified high-level PL code, and its assumed benefit is about $2-3\%$. Lastly, the improvement of "AES-NI (1 x 1)" is a special case, because the cost of memory copies and function calls are considerably reduced by FACE compared

Table 4: Comparison of key-scheduling performances

Platform	Implementation Method	Measurement	Target	Key Size		
				AES-128	AES-192	AES-256
Test Env 1	Table-based	Cycles for key setup	OpenSSL	176	197	259
			This Paper	6067	6134	6276
		Times for key setup	OpenSSL	0.062 µs	0.070 µs	0.091 µs
			This Paper	2.147 µs	2.170 µs	2.221 µs
		Cycle Overhead		5891	5937	6017
		Time Overhead		2.085 µs	2.100 µs	2.130 µs
	Bitsliced	Cycles for key setup	OpenSSL	383	389	478
			This Paper	8901	8908	8992
		Times for key setup	OpenSSL	0.074 µs	0.075 µs	0.083 µs
			This Paper	3.146 µs	3.152 µs	3.177 µs
		Cycle Overhead		8518	8519	8514
		Time Overhead		3.072 µs	3.077 µs	3.094 µs
Test Env 3	Table-based	Cycles for key setup	OpenSSL	121	137	172
			This Paper	3643	3662	3724
		Times for key setup	OpenSSL	0.033 µs	0.036 µs	0.050 µs
			This Paper	0.980 µs	0.993 µs	1.010 µs
		Cycle Overhead		3522	3525	3552
		Time Overhead		0.947 µs	0.957 µs	0.960 µs
	Bitsliced	Cycles for key setup	OpenSSL	242	254	306
			This Paper	5778	5783	5802
		Times for key setup	OpenSSL	0.066 µs	0.069 µs	0.086 µs
			This Paper	1.577 µs	1.591 µs	1.601 µs
		Cycle Overhead		5536	5529	5496
		Time Overhead		1.511 µs	1.522 µs	1.515 µs
	AES-NI	Cycles for key setup	Crypto++	145	173	212
			This Paper (R1)	680	749	783
			This Paper (R2)	7568	7628	7618
		Times for key setup	Crypto++	0.039 µs	0.047 µs	0.058 µs
			This Paper (R1)	0.184 µs	0.196 µs	0.205 µs
			This Paper (R2)	1.994 µs	2.067 µs	2.054 µs
		Cycle Overhead (R1)		535	576	571
		Cycle Overhead (R2)		7428	7455	7406
		Time Overhead (R1)		0.145 µs	0.149 µs	0.147 µs
		Time Overhead (R2)		1.955 µs	2.020 µs	1.996 µs

to 4 x 1. It demonstrates that such operations can be a significant burden.

It seems that the performance improvement is modest, but in effect, such an improvement leads to a difference of hundreds of megabytes per second in throughput. Our experimental results demonstrate that FACE can be applied to any AES CTR implementation, regardless of the implementation method, and can improve performance in various environments.

When applying the technique discussed in section 4.3 or 4.5, our proposed method can create pre-computation values in the initialization stage. This may be an additional burden in terms of efficiency. Therefore, we measure the processing time and cycle counts for the initialization to show that the generation of pre-computation values does not require significant overhead. Table 4 shows the evaluation results of the initialization stage (key scheduling). The key-schedule process of the bitsliced method includes a transformation that converts round keys from a table-based representation to a bitslice representation. All of AES-128, AES-192, and AES-256 require little overhead to generate the temporary lookup table. Thus, the generation of pre-computation values leads to marginal overhead with respect to the performance of FACE.

6 Discussion

As cache-timing attacks have become a promising attacks on software-based AES implementations, they have become significant security threats for AES implementations. Cache-timing attacks on AES exploit the timing variability of data loads from memory while implementations make heavy use of lookup tables. It seems that our proposed

methods are vulnerable to timing attacks because of the use of pre-computed lookup tables. We note that the vulnerabilities caused by timing attacks on FACE are dependent on the adopting implementation method because FACE can be employed in any AES CTR implementation, regardless of implementation method (i.e. table-based, bitsliced, and AES-NI-based). If FACE is applied to the table-based method, the result is also vulnerable to timing attacks. On the other hand, if FACE is applied to the bitsliced method, which offers timing-attack resistance, the result is also cache-timing-attack resistant. It means that the vulnerabilities caused by timing attacks on an AES implementation are not affected by our proposed methods. There are several reasons to explain this.

Several timing attacks against AES make it possible to recover secret information. This is because the indices of the table in the existing table-based implementation are related to secret information. In contrast, FACE adds temporary tables for intermediate values and the indices of our generated tables are independent of secret information. In FACE_{rd0}, FACE_{rd1}, and FACE_{rd2}, the size of the generated table is extremely small and the indices of the tables are fixed. The size of tables are 12 bytes for each FACE_{rd0} and FACE_{rd1}, and 16 bytes for FACE_{rd2}. Modern cache stores groups of bytes in blocks of fixed sizes, called *cache lines*. Common *cache line* sizes are 32 bytes for a Pentium III, and 64 and 128 bytes for more recent processors. Data is transferred between main memory and cache in blocks of *cache line* size. Therefore, all of our generated values in the tables for each phase (FACE_{rd0}, FACE_{rd1}, and FACE_{rd2}) always share a line in the cache on any *cache line* size (*cache hit*). However, the timing variations, which cause information leakage, may exist even if loading occurs only within a fixed cache line. Osvik et al. [OST06] and Bernstein [Ber05] warned that even if secret-dependent accesses are at a finer than cache line granularity, access to different offsets within cache lines may leak information due to cache-bank conflicts. Finally, Yarom et al. [YGH17] demonstrated that the first side-channel attack, which exploits cache-bank conflicts, is feasible on the Sandy Bridge microarchitecture. Such an attack is viable when a secret-dependent access to cache banks exists. But in FACE, the indices of the table lookups are always constant as 0 to 2 in the phase of FACE_{rd0} and FACE_{rd1}, and 0 to 3 in the phase of FACE_{rd2} (all content of tables are used in round operation). In this case, there is no secret-dependent access patterns when FACE accesses the cache. Currently, cache-bank conflicts are no longer an issue since the Haswell processors [Cor18].

In the phase of FACE_{rd1+} or FACE_{rd2+}, our method generates a larger table (1024 bytes for FACE_{rd1+}, 4096 bytes for FACE_{rd2+}). This table might cause a *cache miss*. However, the index of the table is merely a part of a counter value that does not need to be secret (as generally known, keeping the counter value secret adds no security). More precisely, the lookup index is the last byte of the counter, and it even increases linearly. There is no secret-dependent access information while loading data from the table. Furthermore, there is no operation that includes the loaded table value and the secret. In addition, the index is increased by one if the counter is increased by one. Thus, on FACE_{rd1+} phase, *cache miss* caused periodically for every 16 consecutive blocks when only one (64 bytes) cache line is used (caused for every 4 consecutive blocks on FACE_{rd2+} phase). The timing variability of data loads from memory due to *cache miss* would present cyclically, and there is no secret-dependent timing variability in the data loads from the table while processing consecutive blocks for each *cache miss*.

These features show that our approach does not cause additional vulnerabilities to known timing attacks.

7 Conclusion

The AES CTR mode is used in encryption/decryption operations such as streaming services that require high-speed processing because it enables parallel processing. Efforts to improve

the efficiency of AES CTR mode implementation in software have been realized using the bitsliced method. Although performance improvements in algorithms for specialized platforms lack scalability, improvements in the implementation logic of AES can be applied without depending on operating architectures and implementation methods.

This paper proposed FACE, which can improve the performance of the AES CTR mode by using repetitive data. FACE reduces the number of unnecessary calculations during the operation in order to preserve computational resources. To accomplish this, we leverage that very small changes occur in successive blocks in the AES CTR mode, which can be cached and reused. FACE can be employed in any AES CTR implementation, regardless of implementation method (i.e. table-based, bitsliced, and AES-NI-based). As a result, our experimental results are approximately 15%−20% more efficient than those of previously reported methods using a single core. In addition, none of the techniques introduced in this paper are platform specific. Thus, when we apply this technique to implementations of the AES CTR mode, we can expect to achieve performance improvements regardless of the platform or environment. Therefore, FACE provides a computational advantage over high-profile I/O and network services, such as Gigabit ethernet, the Wireless Gigabit Alliance (WiGig) 1.1 [All10] network, and USB 3.0.

We can consider applying our strategy to CAESAR finalist Deoxys [JNPS16], since Deoxys also uses a counter in the tweak input while the plaintext remains unchanged. The caching strategy for Deoxys is slightly more complex than AES because the differences between successive blocks are caused not only by the round transformation but also by the tweakey schedule algorithm. Nonetheless, the additional difference is not considerable. For example, at the end of the first round, the difference is increased only one byte compared to FACE, and there also exists the characteristic of being repeated. It would be interesting to verify whether our caching strategy can be applied to other algorithms that have similar characteristics to the AES CTR mode. This is the subject of our future work to examine the extendability of our caching strategies.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable and helpful comments in improving the quality of this work. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. R7117-16-0161, Anomaly Detection Framework for Autonomous Vehicles).

References

[All08] The Open Mobile Alliance. *DRM Content Format Approved Version 2.1.2*, 2008.

[All10] WiGig Alliance. Defining the future of multi-gigabit wireless communications. *WiGig White Paper*, 2010.

[BBF+02] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient software implementation of aes on 32-bit platforms. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 159–171. Springer, 2002.

[Ber05] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[Bih97] Eli Biham. A fast new des implementation in software. In *International Workshop on Fast Software Encryption*, pages 260–272. Springer, 1997.

[BMN+04] Mark Baugher, D McGrew, M Naslund, E Carrara, and Karl Norrman. The secure real-time transport protocol (SRTP). RFC 3711, 2004.

[BS08] Daniel J Bernstein and Peter Schwabe. New aes software speed records. In *International Conference on Cryptology in India*, pages 322–336. Springer, 2008.

[Cor18] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. Order Number: 248966-040, April 2018.

[CYW03] Francois Charot, Eslam Yahya, and Charles Wagner. Efficient modular-pipelined aes implementation in counter mode on altera fpga. In *International Conference on Field Programmable Logic and Applications*, pages 282–291. Springer, 2003.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[GCVRSPGP10] José M Granado-Criado, Miguel A Vega-Rodríguez, Juan M Sánchez-Pérez, and Juan A Gómez-Pulido. A new methodology to implement the aes algorithm using partial and dynamic reconfiguration. *INTEGRATION, the VLSI journal*, 43(1):72–80, 2010.

[Gue10] Shay Gueron. Intel® advanced encryption standard (AES) new instructions set. *Intel Corporation*, 2010.

[JNPS16] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1.41. *CAESAR Finalists*, October 2016. https://competitions.cr.yp.to/caesar-submissions.html.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 1–17. Springer, 2009.

[LB13] Bin Liu and Bevan M Baas. Parallel aes encryption engines for many-core processor arrays. *IEEE transactions on computers*, 62(3):536–547, 2013.

[Lib] CRYPTO++ Library. http://www.cryptopp.com.

[Mat06] Mitsuru Matsui. How far can we go on the x64 processors? In *International Workshop on Fast Software Encryption*, pages 341–358. Springer, 2006.

[McG02] David McGrew. Counter mode security: Analysis and recommendations. *Cisco Systems, November*, 2:4, 2002.

[MF05] Mitsuru Matsui and Sayaka Fukuda. How to maximize software performance of symmetric primitives on pentium iii and 4 processors. In *International Workshop on Fast Software Encryption*, pages 398–412. Springer, 2005.

[MN07] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 121–134. Springer, 2007.

[MS04] Sumio Morioka and Akashi Satoh. A 10-gbps full-aes crypto design with a twisted bdd s-box architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):686–691, 2004.

[NIS01a] NIST. *Advanced Encryption Standard (AES)*, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NIS01b] NIST. *Recommendation for Block Cipher Modes of Operation (Methods and Techniques)*, 2001. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[NIS04] NIST. *Recommendation for Block Cipher Modes of Operation:The CCM Mode for Authentication and Confidentiality*, 2004. http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf.

[NIS07] NIST. *Recommendation for Block Cipher Modes of Operation:Galois/Counter Mode (GCM) and GMAC*, 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[Pro] The OpenSSL Project. http://www.openssl.org.

[QSH+09] Shanxin Qu, Guochu Shou, Yihong Hu, Zhigang Guo, and Zongjue Qian. High throughput, pipelined implementation of aes on fpga. In *Information Engineering and Electronic Commerce, 2009. IEEC'09. International Symposium on*, pages 542–545. IEEE, 2009.

[RSQL04] Gaël Rouvroy, F-X Standaert, J-J Quisquater, and J-D Legat. Compact and efficient encryption/decryption module for fpga implementation of the aes rijndael very well suited for small embedded applications. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 583–587. IEEE, 2004.

[SRHDP03] Nazar Abbas Saqib, Francisco Rodríguez-Henríquez, and Arturo Díaz-Pérez. Aes algorithm implementation-an efficient approach for sequential and pipeline architectures. In *Computer Science, 2003. ENC 2003. Proceedings of the Fourth Mexican International Conference on*, pages 126–130. IEEE, 2003.

[Wu07] HongJun Wu. *Hongjun's optimized C-code for AES-128 and AES-256.* eSTREAM Project, 2007. http://www.ecrypt.eu.org/stream/svn/viewcvs.cgi/ecrypt/trunk/benchmarks/aes-ctr/aes-128/hongjun/v1/?rev=203#dirlist.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

A Appendix

A.1 Round Transformation Code of Bitsliced FACE

```
.type     __bsaes_face_encrypt8 ,@function
.align    64
__bsaes_face_encrypt8:
          leaq      .LBS0(%rip),%r11
          movdqa    (%rax),%xmm8
          leaq      16(%rax),%rax
          movdqa    80(%r11),%xmm7
          pxor      %xmm8,%xmm15
          pxor      %xmm8,%xmm0
          pxor      %xmm8,%xmm1
          pxor      %xmm8,%xmm2
          pshufb    %xmm7,%xmm15
          pshufb    %xmm7,%xmm0
          pxor      %xmm8,%xmm3
          pxor      %xmm8,%xmm4
          pshufb    %xmm7,%xmm1
          pshufb    %xmm7,%xmm2
          pxor      %xmm8,%xmm5
          pxor      %xmm8,%xmm6
          pshufb    %xmm7,%xmm3
          pshufb    %xmm7,%xmm4
          pshufb    %xmm7,%xmm5
          pshufb    %xmm7,%xmm6
__bsaes_face_encrypt8_bitslice:
          movzbl    44(%rbp), %ecx
          test      %cl, %cl
          jne  .Lctr_face
          movdqa    %xmm15,%xmm0
          movdqa    %xmm15,%xmm1
          paddd     0(%r11),%xmm0
          movdqa    %xmm15,%xmm2
          paddd     16(%r11),%xmm1
          movdqa    %xmm15,%xmm3
          paddd     32(%r11),%xmm2
          movdqa    %xmm15,%xmm4
          paddd     48(%r11),%xmm3
          movdqa    %xmm15,%xmm5
          paddd     64(%r11),%xmm4
          movdqa    %xmm15,%xmm6
          paddd     80(%r11),%xmm5
          paddd     96(%r11),%xmm6
          movdqa    (%rax),%xmm8
          leaq      16(%rax),%rax
          movdqa    −16(%r11),%xmm7
          pxor      %xmm8,%xmm15
          pxor      %xmm8,%xmm0
          pxor      %xmm8,%xmm1
          pxor      %xmm8,%xmm2
          pshufb    %xmm7,%xmm15
          pshufb    %xmm7,%xmm0
          pxor      %xmm8,%xmm3
          pxor      %xmm8,%xmm4
          pshufb    %xmm7,%xmm1
          pshufb    %xmm7,%xmm2
          pxor      %xmm8,%xmm5
          pxor      %xmm8,%xmm6
          pshufb    %xmm7,%xmm3
          pshufb    %xmm7,%xmm4
          pshufb    %xmm7,%xmm5
          pshufb    %xmm7,%xmm6
          leaq      .LBS0(%rip),%r11
          movl      %ebx,%r10d

          movdqa    0(%r11),%xmm7
          movdqa    16(%r11),%xmm8
          movdqa    %xmm5,%xmm9
          psrlq     $1,%xmm5
          movdqa    %xmm3,%xmm10
          psrlq     $1,%xmm3
          pxor      %xmm6,%xmm5
          pxor      %xmm4,%xmm3
          pand      %xmm7,%xmm5
```

```
pand      %xmm7,%xmm3
pxor      %xmm5,%xmm6
psllq     $1,%xmm5
pxor      %xmm3,%xmm4
psllq     $1,%xmm3
pxor      %xmm9,%xmm5
pxor      %xmm10,%xmm3
movdqa    %xmm1,%xmm9
psrlq     $1,%xmm1
movdqa    %xmm15,%xmm10
psrlq     $1,%xmm15
pxor      %xmm2,%xmm1
pxor      %xmm0,%xmm15
pand      %xmm7,%xmm1
pand      %xmm7,%xmm15
pxor      %xmm1,%xmm2
psllq     $1,%xmm1
pxor      %xmm15,%xmm0
psllq     $1,%xmm15
pxor      %xmm9,%xmm1
pxor      %xmm10,%xmm15
movdqa    32(%r11),%xmm7
movdqa    %xmm4,%xmm9
psrlq     $2,%xmm4
movdqa    %xmm3,%xmm10
psrlq     $2,%xmm3
pxor      %xmm6,%xmm4
pxor      %xmm5,%xmm3
pand      %xmm8,%xmm4
pand      %xmm8,%xmm3
pxor      %xmm4,%xmm6
psllq     $2,%xmm4
pxor      %xmm3,%xmm5
psllq     $2,%xmm3
pxor      %xmm9,%xmm4
pxor      %xmm10,%xmm3
movdqa    %xmm0,%xmm9
psrlq     $2,%xmm0
movdqa    %xmm15,%xmm10
psrlq     $2,%xmm15
pxor      %xmm2,%xmm0
pxor      %xmm1,%xmm15
pand      %xmm8,%xmm0
pand      %xmm8,%xmm15
pxor      %xmm0,%xmm2
psllq     $2,%xmm0
pxor      %xmm15,%xmm1
psllq     $2,%xmm15
pxor      %xmm9,%xmm0
pxor      %xmm10,%xmm15
movdqa    %xmm2,%xmm9
psrlq     $4,%xmm2
movdqa    %xmm1,%xmm10
psrlq     $4,%xmm1
pxor      %xmm6,%xmm2
pxor      %xmm5,%xmm1
pand      %xmm7,%xmm2
pand      %xmm7,%xmm1
pxor      %xmm2,%xmm6
psllq     $4,%xmm2
pxor      %xmm1,%xmm5
psllq     $4,%xmm1
pxor      %xmm9,%xmm2
pxor      %xmm10,%xmm1
movdqa    %xmm0,%xmm9
psrlq     $4,%xmm0
movdqa    %xmm15,%xmm10
psrlq     $4,%xmm15
pxor      %xmm4,%xmm0
pxor      %xmm3,%xmm15
pand      %xmm7,%xmm0
pand      %xmm7,%xmm15
```

```
pxor    %xmm0,%xmm4
psllq   $4,%xmm0
pxor    %xmm15,%xmm3
psllq   $4,%xmm15
pxor    %xmm9,%xmm0
pxor    %xmm10,%xmm15
decl    %r10d
decl    %r10d

pxor    %xmm5,%xmm4
pxor    %xmm0,%xmm1
pxor    %xmm15,%xmm2
pxor    %xmm1,%xmm5
pxor    %xmm15,%xmm4
pxor    %xmm2,%xmm5
pxor    %xmm6,%xmm2
pxor    %xmm4,%xmm6
pxor    %xmm3,%xmm2
pxor    %xmm4,%xmm3
pxor    %xmm0,%xmm2
pxor    %xmm6,%xmm1
pxor    %xmm4,%xmm0
movdqa  %xmm6,%xmm10
movdqa  %xmm0,%xmm9
movdqa  %xmm4,%xmm8
movdqa  %xmm1,%xmm12
movdqa  %xmm5,%xmm11
pxor    %xmm3,%xmm10
pxor    %xmm1,%xmm9
pxor    %xmm2,%xmm8
movdqa  %xmm10,%xmm13
pxor    %xmm3,%xmm12
movdqa  %xmm9,%xmm7
pxor    %xmm15,%xmm11
movdqa  %xmm10,%xmm14
por     %xmm8,%xmm9
por     %xmm11,%xmm10
pxor    %xmm7,%xmm14
pand    %xmm11,%xmm13
pxor    %xmm8,%xmm11
pand    %xmm8,%xmm7
pand    %xmm11,%xmm14
movdqa  %xmm2,%xmm11
pxor    %xmm15,%xmm11
pand    %xmm11,%xmm12
pxor    %xmm12,%xmm10
pxor    %xmm12,%xmm9
movdqa  %xmm6,%xmm12
movdqa  %xmm4,%xmm11
pxor    %xmm0,%xmm12
pxor    %xmm5,%xmm11
movdqa  %xmm12,%xmm8
pand    %xmm11,%xmm12
por     %xmm11,%xmm8
pxor    %xmm12,%xmm7
pxor    %xmm14,%xmm10
pxor    %xmm13,%xmm9
pxor    %xmm14,%xmm8
movdqa  %xmm1,%xmm11
pxor    %xmm13,%xmm7
movdqa  %xmm3,%xmm12
pxor    %xmm13,%xmm8
movdqa  %xmm0,%xmm13
pand    %xmm2,%xmm11
movdqa  %xmm6,%xmm14
pand    %xmm15,%xmm12
pand    %xmm4,%xmm13
por     %xmm5,%xmm14
pxor    %xmm11,%xmm10
pxor    %xmm12,%xmm9
pxor    %xmm13,%xmm8
pxor    %xmm14,%xmm7
movdqa  %xmm10,%xmm11
pand    %xmm8,%xmm10
pxor    %xmm9,%xmm11
movdqa  %xmm7,%xmm13
movdqa  %xmm11,%xmm14
pxor    %xmm10,%xmm13
pand    %xmm13,%xmm14
movdqa  %xmm8,%xmm12
pxor    %xmm9,%xmm14
pxor    %xmm7,%xmm12
pxor    %xmm9,%xmm10
pand    %xmm10,%xmm12
movdqa  %xmm13,%xmm9
pxor    %xmm7,%xmm12
pxor    %xmm12,%xmm9
pxor    %xmm12,%xmm8
pand    %xmm7,%xmm9
pxor    %xmm9,%xmm13
pxor    %xmm9,%xmm8
pand    %xmm14,%xmm13
pxor    %xmm11,%xmm13
movdqa  %xmm5,%xmm11
movdqa  %xmm4,%xmm7
movdqa  %xmm14,%xmm9
pxor    %xmm13,%xmm9
pand    %xmm5,%xmm9
pxor    %xmm4,%xmm5
pand    %xmm14,%xmm4
pand    %xmm13,%xmm5
pxor    %xmm4,%xmm5
pxor    %xmm9,%xmm4
pxor    %xmm15,%xmm11
pxor    %xmm2,%xmm7
pxor    %xmm12,%xmm14
pxor    %xmm8,%xmm13
movdqa  %xmm14,%xmm10
movdqa  %xmm12,%xmm9
pxor    %xmm13,%xmm10
pxor    %xmm8,%xmm9
pand    %xmm11,%xmm10
pand    %xmm15,%xmm9
pxor    %xmm7,%xmm11
pxor    %xmm2,%xmm15
pand    %xmm14,%xmm7
pand    %xmm12,%xmm2
pand    %xmm13,%xmm11
pand    %xmm8,%xmm15
pxor    %xmm11,%xmm7
pxor    %xmm2,%xmm15
pxor    %xmm10,%xmm11
pxor    %xmm9,%xmm2
pxor    %xmm11,%xmm5
pxor    %xmm11,%xmm15
pxor    %xmm7,%xmm4
pxor    %xmm7,%xmm2
movdqa  %xmm6,%xmm11
movdqa  %xmm0,%xmm7
pxor    %xmm3,%xmm11
pxor    %xmm1,%xmm7
movdqa  %xmm14,%xmm10
movdqa  %xmm12,%xmm9
pxor    %xmm13,%xmm10
pxor    %xmm8,%xmm9
pand    %xmm11,%xmm10
pand    %xmm3,%xmm9
pxor    %xmm7,%xmm11
pxor    %xmm1,%xmm3
pand    %xmm14,%xmm7
pand    %xmm12,%xmm1
pand    %xmm13,%xmm11
pand    %xmm8,%xmm3
pxor    %xmm11,%xmm7
pxor    %xmm1,%xmm3
pxor    %xmm10,%xmm11
pxor    %xmm9,%xmm1
pxor    %xmm12,%xmm14
pxor    %xmm8,%xmm13
movdqa  %xmm14,%xmm10
pxor    %xmm13,%xmm10
pand    %xmm6,%xmm10
pxor    %xmm0,%xmm6
pand    %xmm14,%xmm0
pand    %xmm13,%xmm6
pxor    %xmm0,%xmm6
```

```
        pxor    %xmm10,%xmm0
        pxor    %xmm11,%xmm6
        pxor    %xmm11,%xmm3
        pxor    %xmm7,%xmm0
        pxor    %xmm7,%xmm1
        pxor    %xmm15,%xmm6
        pxor    %xmm5,%xmm0
        pxor    %xmm6,%xmm3
        pxor    %xmm15,%xmm5
        pxor    %xmm0,%xmm15
        pxor    %xmm4,%xmm0
        pxor    %xmm1,%xmm4
        pxor    %xmm2,%xmm1
        pxor    %xmm4,%xmm2
        pxor    %xmm4,%xmm3
        pxor    %xmm2,%xmm5

        pshufd  $147,%xmm15,%xmm7
        pshufd  $147,%xmm0,%xmm8
        pxor    %xmm7,%xmm15
        pshufd  $147,%xmm3,%xmm9
        pxor    %xmm8,%xmm0
        pshufd  $147,%xmm5,%xmm10
        pxor    %xmm9,%xmm3
        pshufd  $147,%xmm2,%xmm11
        pxor    %xmm10,%xmm5
        pshufd  $147,%xmm6,%xmm12
        pxor    %xmm11,%xmm2
        pshufd  $147,%xmm1,%xmm13
        pxor    %xmm12,%xmm6
        pshufd  $147,%xmm4,%xmm14
        pxor    %xmm13,%xmm1
        pxor    %xmm14,%xmm4
        pxor    %xmm15,%xmm8
        pxor    %xmm4,%xmm7
        pxor    %xmm4,%xmm8
        pshufd  $78,%xmm15,%xmm15
        pxor    %xmm0,%xmm9
        pshufd  $78,%xmm0,%xmm0
        pxor    %xmm2,%xmm12
        pxor    %xmm7,%xmm15
        pxor    %xmm6,%xmm13
        pxor    %xmm8,%xmm0
        pxor    %xmm5,%xmm11
        pshufd  $78,%xmm2,%xmm7
        pxor    %xmm1,%xmm14
        pshufd  $78,%xmm6,%xmm8
        pxor    %xmm3,%xmm10
        pshufd  $78,%xmm5,%xmm2
        pxor    %xmm4,%xmm10
        pshufd  $78,%xmm4,%xmm6
        pxor    %xmm4,%xmm11
        pshufd  $78,%xmm1,%xmm5
        pxor    %xmm11,%xmm7
        pshufd  $78,%xmm3,%xmm1
        pxor    %xmm12,%xmm8
        pxor    %xmm10,%xmm2
        pxor    %xmm14,%xmm6
        pxor    %xmm13,%xmm5
        movdqa  %xmm7,%xmm3
        pxor    %xmm9,%xmm1
        movdqa  %xmm8,%xmm4
        movdqa  48(%r11),%xmm7
        jnz     .
             Lenc_addroundkey_forcache
        movdqa  64(%r11),%xmm7
        jmp     .
             Lenc_addroundkey_forcache

.Lenc_addroundkey_forcache:
        pxor    0(%rax),%xmm15
        pxor    16(%rax),%xmm0
        pxor    32(%rax),%xmm1
        pxor    48(%rax),%xmm2
        pshufb  %xmm7,%xmm15
        pshufb  %xmm7,%xmm0
        pxor    64(%rax),%xmm3
        pxor    80(%rax),%xmm4
        pshufb  %xmm7,%xmm1
        pshufb  %xmm7,%xmm2
        pxor    96(%rax),%xmm5
        pxor    112(%rax),%xmm6
        pshufb  %xmm7,%xmm3
        pshufb  %xmm7,%xmm4
        pshufb  %xmm7,%xmm5
        pshufb  %xmm7,%xmm6
        leaq    128(%rax),%rax
        decl    %r10d

        pxor    %xmm5,%xmm4
        pxor    %xmm0,%xmm1
        pxor    %xmm15,%xmm2
        pxor    %xmm1,%xmm5
        pxor    %xmm15,%xmm4
        pxor    %xmm2,%xmm5
        pxor    %xmm6,%xmm2
        pxor    %xmm4,%xmm6
        pxor    %xmm3,%xmm2
        pxor    %xmm4,%xmm3
        pxor    %xmm0,%xmm2
        pxor    %xmm6,%xmm1
        pxor    %xmm4,%xmm0
        movdqa  %xmm6,%xmm10
        movdqa  %xmm0,%xmm9
        movdqa  %xmm4,%xmm8
        movdqa  %xmm1,%xmm12
        movdqa  %xmm5,%xmm11
        pxor    %xmm3,%xmm10
        pxor    %xmm1,%xmm9
        pxor    %xmm2,%xmm8
        movdqa  %xmm10,%xmm13
        pxor    %xmm3,%xmm12
        movdqa  %xmm9,%xmm7
        pxor    %xmm15,%xmm11
        movdqa  %xmm10,%xmm14
        por     %xmm8,%xmm9
        por     %xmm11,%xmm10
        pxor    %xmm7,%xmm14
        pand    %xmm11,%xmm13
        pxor    %xmm8,%xmm11
        pand    %xmm8,%xmm7
        pand    %xmm11,%xmm14
        movdqa  %xmm2,%xmm11
        pxor    %xmm15,%xmm11
        pand    %xmm11,%xmm12
        pxor    %xmm12,%xmm10
        pxor    %xmm12,%xmm9
        movdqa  %xmm6,%xmm12
        movdqa  %xmm4,%xmm11
        pxor    %xmm0,%xmm12
        pxor    %xmm5,%xmm11
        movdqa  %xmm12,%xmm8
        pand    %xmm11,%xmm12
        por     %xmm11,%xmm8
        pxor    %xmm12,%xmm7
        pxor    %xmm14,%xmm10
        pxor    %xmm13,%xmm9
        pxor    %xmm14,%xmm8
        movdqa  %xmm3,%xmm12
        pxor    %xmm13,%xmm7
        movdqa  %xmm0,%xmm13
        pand    %xmm2,%xmm11
        movdqa  %xmm6,%xmm14
        pand    %xmm15,%xmm12
        pand    %xmm4,%xmm13
        por     %xmm5,%xmm14
        pxor    %xmm11,%xmm10
        pxor    %xmm12,%xmm9
        pxor    %xmm13,%xmm8
        pxor    %xmm14,%xmm7
        movdqa  %xmm10,%xmm11
        pand    %xmm8,%xmm10
        pxor    %xmm9,%xmm11
        movdqa  %xmm7,%xmm13
```

```
movdqa  %xmm11,%xmm14
pxor    %xmm10,%xmm13
pand    %xmm13,%xmm14
movdqa  %xmm8,%xmm12
pxor    %xmm9,%xmm14
pxor    %xmm7,%xmm12
pxor    %xmm9,%xmm10
pand    %xmm10,%xmm12
movdqa  %xmm13,%xmm9
pxor    %xmm7,%xmm12
pxor    %xmm12,%xmm9
pxor    %xmm12,%xmm8
pand    %xmm7,%xmm9
pxor    %xmm9,%xmm13
pxor    %xmm9,%xmm8
pand    %xmm14,%xmm13
pxor    %xmm11,%xmm13
movdqa  %xmm5,%xmm11
movdqa  %xmm4,%xmm7
movdqa  %xmm14,%xmm9
pxor    %xmm13,%xmm9
pand    %xmm5,%xmm9
pxor    %xmm4,%xmm5
pand    %xmm14,%xmm4
pand    %xmm13,%xmm5
pxor    %xmm4,%xmm5
pxor    %xmm9,%xmm4
pxor    %xmm15,%xmm11
pxor    %xmm2,%xmm7
pxor    %xmm12,%xmm14
pxor    %xmm8,%xmm13
movdqa  %xmm14,%xmm10
movdqa  %xmm12,%xmm9
pxor    %xmm13,%xmm10
pxor    %xmm8,%xmm9
pand    %xmm11,%xmm10
pand    %xmm15,%xmm9
pxor    %xmm7,%xmm11
pxor    %xmm2,%xmm15
pand    %xmm14,%xmm7
pand    %xmm12,%xmm2
pand    %xmm13,%xmm11
pand    %xmm8,%xmm15
pxor    %xmm11,%xmm7
pxor    %xmm2,%xmm15
pxor    %xmm10,%xmm11
pxor    %xmm9,%xmm2
pxor    %xmm11,%xmm5
pxor    %xmm11,%xmm15
pxor    %xmm7,%xmm4
pxor    %xmm7,%xmm2
movdqa  %xmm6,%xmm11
movdqa  %xmm0,%xmm7
pxor    %xmm3,%xmm11
pxor    %xmm1,%xmm7
movdqa  %xmm14,%xmm10
movdqa  %xmm12,%xmm9
pxor    %xmm13,%xmm10
pxor    %xmm8,%xmm9
pand    %xmm11,%xmm10
pand    %xmm3,%xmm9
pxor    %xmm7,%xmm11
pxor    %xmm1,%xmm3
pand    %xmm14,%xmm7
pand    %xmm12,%xmm1
pand    %xmm13,%xmm11
pand    %xmm8,%xmm3
pxor    %xmm11,%xmm7
pxor    %xmm1,%xmm3
pxor    %xmm10,%xmm11
pxor    %xmm9,%xmm1
pxor    %xmm12,%xmm14
pxor    %xmm8,%xmm13
movdqa  %xmm14,%xmm10
pxor    %xmm13,%xmm10
pand    %xmm6,%xmm10
pxor    %xmm0,%xmm6
pand    %xmm14,%xmm0
```

```
pand    %xmm13,%xmm6
pxor    %xmm0,%xmm6
pxor    %xmm10,%xmm0
pxor    %xmm11,%xmm6
pxor    %xmm11,%xmm3
pxor    %xmm7,%xmm0
pxor    %xmm7,%xmm1
pxor    %xmm15,%xmm6
pxor    %xmm5,%xmm0
pxor    %xmm6,%xmm3
pxor    %xmm15,%xmm5
pxor    %xmm0,%xmm15
pxor    %xmm4,%xmm0
pxor    %xmm1,%xmm4
pxor    %xmm2,%xmm1
pxor    %xmm4,%xmm2
pxor    %xmm4,%xmm3
pxor    %xmm2,%xmm5

pshufd  $147,%xmm15,%xmm7
pshufd  $147,%xmm0,%xmm8
pxor    %xmm7,%xmm15
pshufd  $147,%xmm3,%xmm9
pxor    %xmm8,%xmm0
pshufd  $147,%xmm5,%xmm10
pxor    %xmm9,%xmm3
pshufd  $147,%xmm2,%xmm11
pxor    %xmm10,%xmm5
pshufd  $147,%xmm6,%xmm12
pxor    %xmm11,%xmm2
pshufd  $147,%xmm1,%xmm13
pxor    %xmm12,%xmm6
pshufd  $147,%xmm4,%xmm14
pxor    %xmm13,%xmm1
pxor    %xmm14,%xmm4
pxor    %xmm15,%xmm8
pxor    %xmm4,%xmm7
pxor    %xmm4,%xmm8
pshufd  $78,%xmm15,%xmm15
pxor    %xmm0,%xmm9
pshufd  $78,%xmm0,%xmm0
pxor    %xmm2,%xmm12
pxor    %xmm7,%xmm15
pxor    %xmm6,%xmm13
pxor    %xmm8,%xmm0
pxor    %xmm5,%xmm11
pshufd  $78,%xmm2,%xmm7
pxor    %xmm1,%xmm14
pshufd  $78,%xmm6,%xmm8
pxor    %xmm3,%xmm10
pshufd  $78,%xmm5,%xmm2
pxor    %xmm4,%xmm10
pshufd  $78,%xmm4,%xmm6
pxor    %xmm4,%xmm11
pshufd  $78,%xmm1,%xmm5
pxor    %xmm11,%xmm7
pshufd  $78,%xmm3,%xmm1
pxor    %xmm12,%xmm8
pxor    %xmm10,%xmm2
pxor    %xmm14,%xmm6
pxor    %xmm13,%xmm5
movdqa  %xmm7,%xmm3
pxor    %xmm9,%xmm1
movdqa  %xmm8,%xmm4
movdqa  48(%r11),%xmm7

pxor    0(%rax),%xmm15
pxor    16(%rax),%xmm0
pxor    32(%rax),%xmm1
pxor    48(%rax),%xmm2
pshufb  %xmm7,%xmm15
pshufb  %xmm7,%xmm0
pxor    64(%rax),%xmm3
pxor    80(%rax),%xmm4
pshufb  %xmm7,%xmm1
pshufb  %xmm7,%xmm2
pxor    96(%rax),%xmm5
pxor    112(%rax),%xmm6
```

```
        pshufb  %xmm7,%xmm3
        pshufb  %xmm7,%xmm4
        pshufb  %xmm7,%xmm5
        pshufb  %xmm7,%xmm6
        leaq    128(%rax),%rax

        movdqu  248(%r15),%xmm8
        movdqa  %xmm15,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4344(%r15)
        movdqu  264(%r15),%xmm8
        movdqa  %xmm0,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4360(%r15)
        movdqu  280(%r15),%xmm8
        movdqa  %xmm1,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4376(%r15)
        movdqu  296(%r15),%xmm8
        movdqa  %xmm2,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4392(%r15)
        movdqu  312(%r15),%xmm8
        movdqa  %xmm3,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4408(%r15)
        movdqu  328(%r15),%xmm8
        movdqa  %xmm4,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4424(%r15)
        movdqu  344(%r15),%xmm8
        movdqa  %xmm5,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4440(%r15)
        movdqu  360(%r15),%xmm8
        movdqa  %xmm6,%xmm9
        pxor    %xmm8,%xmm9
        movdqu  %xmm9, 4456(%r15)
        decl    %r10d
        jmp     .Lenc_sbox

.align  16
.Lctr_face:
        leaq    .LBS0(%rip),%r11
        movl    %ebx,%r10d
        leaq    248(%r15),%r8
        shl     $4, %ecx
        add     %rcx,%r8
        leaq    4344(%r15),%r9
        movdqu  (%r9),%xmm15
        movdqu  16(%r9),%xmm0
        movdqu  32(%r9),%xmm1
        movdqu  48(%r9),%xmm2
        movdqu  64(%r9),%xmm3
        movdqu  80(%r9),%xmm4
        movdqu  96(%r9),%xmm5
        movdqu  112(%r9),%xmm6
        movdqu  (%r8),%xmm8
        pxor    %xmm8,%xmm15
        movdqu  16(%r8),%xmm8
        pxor    %xmm8,%xmm0
        movdqu  32(%r8),%xmm8
        pxor    %xmm8,%xmm1
        movdqu  48(%r8),%xmm8
        pxor    %xmm8,%xmm2
        movdqu  64(%r8),%xmm8
        pxor    %xmm8,%xmm3
        movdqu  80(%r8),%xmm8
        pxor    %xmm8,%xmm4
        movdqu  96(%r8),%xmm8
        pxor    %xmm8,%xmm5
        movdqu  112(%r8),%xmm8
        pxor    %xmm8,%xmm6
        leaq    272(%rax),%rax
        decl    %r10d
        decl    %r10d
        decl    %r10d
        decl    %r10d

.align  16
.Lenc_sbox:
        pxor    %xmm5,%xmm4
        pxor    %xmm0,%xmm1
        pxor    %xmm15,%xmm2
        pxor    %xmm1,%xmm5
        pxor    %xmm15,%xmm4
        pxor    %xmm2,%xmm5
        pxor    %xmm6,%xmm2
        pxor    %xmm4,%xmm6
        pxor    %xmm3,%xmm2
        pxor    %xmm4,%xmm3
        pxor    %xmm0,%xmm2
        pxor    %xmm6,%xmm1
        pxor    %xmm4,%xmm0
        movdqa  %xmm6,%xmm10
        movdqa  %xmm0,%xmm9
        movdqa  %xmm4,%xmm8
        movdqa  %xmm1,%xmm12
        movdqa  %xmm5,%xmm11
        pxor    %xmm3,%xmm10
        pxor    %xmm1,%xmm9
        pxor    %xmm2,%xmm8
        movdqa  %xmm10,%xmm13
        pxor    %xmm3,%xmm12
        movdqa  %xmm9,%xmm7
        pxor    %xmm15,%xmm11
        movdqa  %xmm10,%xmm14
        por     %xmm8,%xmm9
        por     %xmm11,%xmm10
        pxor    %xmm7,%xmm14
        pand    %xmm11,%xmm13
        pxor    %xmm8,%xmm11
        pand    %xmm8,%xmm7
        pand    %xmm11,%xmm14
        movdqa  %xmm2,%xmm11
        pxor    %xmm15,%xmm11
        pand    %xmm11,%xmm12
        pxor    %xmm12,%xmm10
        pxor    %xmm12,%xmm9
        movdqa  %xmm6,%xmm12
        movdqa  %xmm4,%xmm11
        pxor    %xmm0,%xmm12
        pxor    %xmm5,%xmm11
        movdqa  %xmm12,%xmm8
        pand    %xmm11,%xmm12
        por     %xmm11,%xmm8
        pxor    %xmm12,%xmm7
        pxor    %xmm14,%xmm10
        pxor    %xmm13,%xmm9
        pxor    %xmm14,%xmm8
        movdqa  %xmm1,%xmm11
        pxor    %xmm13,%xmm7
        movdqa  %xmm3,%xmm12
        pxor    %xmm13,%xmm8
        movdqa  %xmm0,%xmm13
        pand    %xmm2,%xmm11
        movdqa  %xmm6,%xmm14
        pand    %xmm15,%xmm12
        pand    %xmm4,%xmm13
        por     %xmm5,%xmm14
        pxor    %xmm11,%xmm10
        pxor    %xmm12,%xmm9
        pxor    %xmm13,%xmm8
        pxor    %xmm14,%xmm7
        movdqa  %xmm10,%xmm11
        pand    %xmm8,%xmm10
        pxor    %xmm9,%xmm11
        movdqa  %xmm7,%xmm13
        movdqa  %xmm11,%xmm14
        pxor    %xmm10,%xmm13
        pand    %xmm13,%xmm14
        movdqa  %xmm8,%xmm12
        pxor    %xmm9,%xmm14
        pxor    %xmm7,%xmm12
        pxor    %xmm9,%xmm10
        pand    %xmm10,%xmm12
```

```
        movdqa  %xmm13,%xmm9
        pxor    %xmm7,%xmm12
        pxor    %xmm12,%xmm9
        pxor    %xmm12,%xmm8
        pand    %xmm7,%xmm9
        pxor    %xmm9,%xmm13
        pxor    %xmm9,%xmm8
        pand    %xmm14,%xmm13
        pxor    %xmm11,%xmm13
        movdqa  %xmm5,%xmm11
        movdqa  %xmm4,%xmm7
        movdqa  %xmm14,%xmm9
        pxor    %xmm13,%xmm9
        pand    %xmm5,%xmm9
        pxor    %xmm4,%xmm5
        pand    %xmm14,%xmm4
        pand    %xmm13,%xmm5
        pxor    %xmm4,%xmm5
        pxor    %xmm9,%xmm4
        pxor    %xmm15,%xmm11
        pxor    %xmm2,%xmm7
        pxor    %xmm12,%xmm14
        pxor    %xmm8,%xmm13
        movdqa  %xmm14,%xmm10
        movdqa  %xmm12,%xmm9
        pxor    %xmm13,%xmm10
        pxor    %xmm8,%xmm9
        pand    %xmm11,%xmm10
        pand    %xmm15,%xmm9
        pxor    %xmm7,%xmm11
        pxor    %xmm2,%xmm15
        pand    %xmm14,%xmm7
        pand    %xmm12,%xmm2
        pand    %xmm13,%xmm11
        pand    %xmm8,%xmm15
        pxor    %xmm11,%xmm7
        pxor    %xmm2,%xmm15
        pxor    %xmm10,%xmm11
        pxor    %xmm9,%xmm2
        pxor    %xmm11,%xmm5
        pxor    %xmm11,%xmm15
        pxor    %xmm7,%xmm4
        pxor    %xmm7,%xmm2
        movdqa  %xmm6,%xmm11
        movdqa  %xmm0,%xmm7
        pxor    %xmm3,%xmm11
        pxor    %xmm1,%xmm7
        movdqa  %xmm14,%xmm10
        movdqa  %xmm12,%xmm9
        pxor    %xmm13,%xmm10
        pxor    %xmm8,%xmm9
        pand    %xmm11,%xmm10
        pand    %xmm3,%xmm9
        pxor    %xmm7,%xmm11
        pxor    %xmm1,%xmm3
        pand    %xmm14,%xmm7
        pand    %xmm12,%xmm1
        pand    %xmm13,%xmm11
        pand    %xmm8,%xmm3
        pxor    %xmm11,%xmm7
        pxor    %xmm1,%xmm3
        pxor    %xmm10,%xmm11
        pxor    %xmm9,%xmm1
        pxor    %xmm12,%xmm14
        pxor    %xmm8,%xmm13
        movdqa  %xmm14,%xmm10
        pxor    %xmm13,%xmm10
        pand    %xmm6,%xmm10
        pxor    %xmm0,%xmm6
        pand    %xmm14,%xmm0
        pand    %xmm13,%xmm6
        pxor    %xmm0,%xmm6
        pxor    %xmm10,%xmm0
        pxor    %xmm11,%xmm6
        pxor    %xmm11,%xmm3
        pxor    %xmm7,%xmm0
        pxor    %xmm7,%xmm1
        pxor    %xmm15,%xmm6
        pxor    %xmm5,%xmm0
        pxor    %xmm6,%xmm3
        pxor    %xmm15,%xmm5
        pxor    %xmm0,%xmm15
        pxor    %xmm4,%xmm0
        pxor    %xmm1,%xmm4
        pxor    %xmm2,%xmm1
        pxor    %xmm4,%xmm2
        pxor    %xmm4,%xmm3
        pxor    %xmm2,%xmm5

        pshufd  $147,%xmm15,%xmm7
        pshufd  $147,%xmm0,%xmm8
        pxor    %xmm7,%xmm15
        pshufd  $147,%xmm3,%xmm9
        pxor    %xmm8,%xmm0
        pshufd  $147,%xmm5,%xmm10
        pxor    %xmm9,%xmm3
        pshufd  $147,%xmm2,%xmm11
        pxor    %xmm10,%xmm5
        pshufd  $147,%xmm6,%xmm12
        pxor    %xmm11,%xmm2
        pshufd  $147,%xmm1,%xmm13
        pxor    %xmm12,%xmm6
        pshufd  $147,%xmm4,%xmm14
        pxor    %xmm13,%xmm1
        pxor    %xmm14,%xmm4
        pxor    %xmm15,%xmm8
        pxor    %xmm4,%xmm7
        pxor    %xmm4,%xmm8
        pshufd  $78,%xmm15,%xmm15
        pxor    %xmm0,%xmm9
        pshufd  $78,%xmm0,%xmm0
        pxor    %xmm2,%xmm12
        pxor    %xmm7,%xmm15
        pxor    %xmm6,%xmm13
        pxor    %xmm8,%xmm0
        pxor    %xmm5,%xmm11
        pshufd  $78,%xmm2,%xmm7
        pxor    %xmm1,%xmm14
        pshufd  $78,%xmm6,%xmm8
        pxor    %xmm3,%xmm10
        pshufd  $78,%xmm5,%xmm2
        pxor    %xmm4,%xmm10
        pshufd  $78,%xmm4,%xmm6
        pxor    %xmm4,%xmm11
        pshufd  $78,%xmm1,%xmm5
        pxor    %xmm11,%xmm7
        pshufd  $78,%xmm3,%xmm1
        pxor    %xmm12,%xmm8
        pxor    %xmm10,%xmm2
        pxor    %xmm14,%xmm6
        pxor    %xmm13,%xmm5
        movdqa  %xmm7,%xmm3
        pxor    %xmm9,%xmm1
        movdqa  %xmm8,%xmm4
        movdqa  48(%r11),%xmm7
        jnz     .Lenc_addroundkey
        movdqa  64(%r11),%xmm7
        jmp     .Lenc_addroundkey

.Lenc_addroundkey:
        pxor    0(%rax),%xmm15
        pxor    16(%rax),%xmm0
        pxor    32(%rax),%xmm1
        pxor    48(%rax),%xmm2
        pshufb  %xmm7,%xmm15
        pshufb  %xmm7,%xmm0
        pxor    64(%rax),%xmm3
        pxor    80(%rax),%xmm4
        pshufb  %xmm7,%xmm1
        pshufb  %xmm7,%xmm2
        pxor    96(%rax),%xmm5
        pxor    112(%rax),%xmm6
        pshufb  %xmm7,%xmm3
        pshufb  %xmm7,%xmm4
        pshufb  %xmm7,%xmm5
        pshufb  %xmm7,%xmm6
```

```
        leaq    128(%rax),%rax
        decl    %r10d
        jl      .Lenc_done
        jmp .Lenc_sbox

.align  16
.Lenc_done:
        pxor    %xmm5,%xmm4
        pxor    %xmm0,%xmm1
        pxor    %xmm15,%xmm2
        pxor    %xmm1,%xmm5
        pxor    %xmm15,%xmm4
        pxor    %xmm2,%xmm5
        pxor    %xmm6,%xmm2
        pxor    %xmm4,%xmm6
        pxor    %xmm3,%xmm2
        pxor    %xmm4,%xmm3
        pxor    %xmm0,%xmm2
        pxor    %xmm6,%xmm1
        pxor    %xmm4,%xmm0
        movdqa  %xmm6,%xmm10
        movdqa  %xmm0,%xmm9
        movdqa  %xmm4,%xmm8
        movdqa  %xmm1,%xmm12
        movdqa  %xmm5,%xmm11
        pxor    %xmm3,%xmm10
        pxor    %xmm1,%xmm9
        pxor    %xmm2,%xmm8
        movdqa  %xmm10,%xmm13
        pxor    %xmm3,%xmm12
        movdqa  %xmm9,%xmm7
        pxor    %xmm15,%xmm11
        movdqa  %xmm10,%xmm14
        por     %xmm8,%xmm9
        por     %xmm11,%xmm10
        pxor    %xmm7,%xmm14
        pand    %xmm11,%xmm13
        pxor    %xmm8,%xmm11
        pand    %xmm8,%xmm7
        pand    %xmm11,%xmm14
        movdqa  %xmm2,%xmm11
        pxor    %xmm15,%xmm11
        pand    %xmm11,%xmm12
        pxor    %xmm12,%xmm10
        pxor    %xmm12,%xmm9
        movdqa  %xmm6,%xmm12
        movdqa  %xmm4,%xmm11
        pxor    %xmm0,%xmm12
        pxor    %xmm5,%xmm11
        movdqa  %xmm12,%xmm8
        pand    %xmm11,%xmm12
        por     %xmm11,%xmm8
        pxor    %xmm12,%xmm7
        pxor    %xmm14,%xmm10
        pxor    %xmm13,%xmm9
        pxor    %xmm14,%xmm8
        movdqa  %xmm1,%xmm11
        pxor    %xmm13,%xmm7
        movdqa  %xmm3,%xmm12
        pxor    %xmm13,%xmm8
        movdqa  %xmm0,%xmm13
        pand    %xmm2,%xmm11
        movdqa  %xmm6,%xmm14
        pand    %xmm15,%xmm12
        pand    %xmm4,%xmm13
        por     %xmm5,%xmm14
        pxor    %xmm11,%xmm10
        pxor    %xmm12,%xmm9
        pxor    %xmm13,%xmm8
        pxor    %xmm14,%xmm7
        movdqa  %xmm10,%xmm11
        pand    %xmm8,%xmm10
        pxor    %xmm9,%xmm11
        movdqa  %xmm7,%xmm13
        movdqa  %xmm11,%xmm14
        pxor    %xmm10,%xmm13
        pand    %xmm13,%xmm14
        movdqa  %xmm8,%xmm12
        pxor    %xmm9,%xmm14
        pxor    %xmm7,%xmm12
        pxor    %xmm9,%xmm10
        pand    %xmm10,%xmm12
        movdqa  %xmm13,%xmm9
        pxor    %xmm7,%xmm12
        pxor    %xmm12,%xmm9
        pxor    %xmm12,%xmm8
        pand    %xmm7,%xmm9
        pxor    %xmm9,%xmm13
        pxor    %xmm9,%xmm8
        pand    %xmm14,%xmm13
        pxor    %xmm11,%xmm13
        movdqa  %xmm5,%xmm11
        movdqa  %xmm4,%xmm7
        movdqa  %xmm14,%xmm9
        pxor    %xmm13,%xmm9
        pand    %xmm5,%xmm9
        pxor    %xmm4,%xmm5
        pand    %xmm14,%xmm4
        pand    %xmm13,%xmm5
        pxor    %xmm4,%xmm5
        pxor    %xmm9,%xmm4
        pxor    %xmm15,%xmm11
        pxor    %xmm2,%xmm7
        pxor    %xmm12,%xmm14
        pxor    %xmm8,%xmm13
        movdqa  %xmm14,%xmm10
        movdqa  %xmm12,%xmm9
        pxor    %xmm13,%xmm10
        pxor    %xmm8,%xmm9
        pand    %xmm11,%xmm10
        pand    %xmm3,%xmm9
        pxor    %xmm7,%xmm11
        pxor    %xmm1,%xmm3
        pand    %xmm14,%xmm7
        pand    %xmm12,%xmm1
        pand    %xmm13,%xmm11
        pand    %xmm8,%xmm3
        pxor    %xmm11,%xmm7
        pxor    %xmm1,%xmm3
        pxor    %xmm10,%xmm11
        pxor    %xmm9,%xmm1
        pxor    %xmm12,%xmm14
        pxor    %xmm8,%xmm13
        movdqa  %xmm14,%xmm10
        pxor    %xmm13,%xmm10
        pand    %xmm6,%xmm10
        pxor    %xmm0,%xmm6
        pand    %xmm14,%xmm0
        pand    %xmm13,%xmm6
        pxor    %xmm0,%xmm6
        pxor    %xmm10,%xmm0
        pxor    %xmm11,%xmm6
```

```
        pxor    %xmm11,%xmm3
        pxor    %xmm7,%xmm0
        pxor    %xmm7,%xmm1
        pxor    %xmm15,%xmm6
        pxor    %xmm5,%xmm0
        pxor    %xmm6,%xmm3
        pxor    %xmm15,%xmm5
        pxor    %xmm0,%xmm15
        pxor    %xmm4,%xmm0
        pxor    %xmm1,%xmm4
        pxor    %xmm2,%xmm1
        pxor    %xmm4,%xmm2
        pxor    %xmm4,%xmm3
        pxor    %xmm2,%xmm5

        movdqa  0(%r11),%xmm7
        movdqa  16(%r11),%xmm8
        movdqa  %xmm1,%xmm9
        psrlq   $1,%xmm1
        movdqa  %xmm2,%xmm10
        psrlq   $1,%xmm2
        pxor    %xmm4,%xmm1
        pxor    %xmm6,%xmm2
        pand    %xmm7,%xmm1
        pand    %xmm7,%xmm2
        pxor    %xmm1,%xmm4
        psllq   $1,%xmm1
        pxor    %xmm2,%xmm6
        psllq   $1,%xmm2
        pxor    %xmm9,%xmm1
        pxor    %xmm10,%xmm2
        movdqa  %xmm3,%xmm9
        psrlq   $1,%xmm3
        movdqa  %xmm15,%xmm10
        psrlq   $1,%xmm15
        pxor    %xmm5,%xmm3
        pxor    %xmm0,%xmm15
        pand    %xmm7,%xmm3
        pand    %xmm7,%xmm15
        pxor    %xmm3,%xmm5
        psllq   $1,%xmm3
        pxor    %xmm15,%xmm0
        psllq   $1,%xmm15
        pxor    %xmm9,%xmm3
        pxor    %xmm10,%xmm15
        movdqa  32(%r11),%xmm7
        movdqa  %xmm6,%xmm9
        psrlq   $2,%xmm6
        movdqa  %xmm2,%xmm10
        psrlq   $2,%xmm2
        pxor    %xmm4,%xmm6
        pxor    %xmm1,%xmm2
        pand    %xmm8,%xmm6
        pand    %xmm8,%xmm2
        pxor    %xmm6,%xmm4
        psllq   $2,%xmm6
        pxor    %xmm2,%xmm1
        psllq   $2,%xmm2
        pxor    %xmm9,%xmm6
        pxor    %xmm10,%xmm2
        movdqa  %xmm0,%xmm9
        psrlq   $2,%xmm0
        movdqa  %xmm15,%xmm10
        psrlq   $2,%xmm15
        pxor    %xmm5,%xmm0
        pxor    %xmm3,%xmm15
        pand    %xmm8,%xmm0
        pand    %xmm8,%xmm15
        pxor    %xmm0,%xmm5
        psllq   $2,%xmm0
        pxor    %xmm15,%xmm3
        psllq   $2,%xmm15
        pxor    %xmm9,%xmm0
        pxor    %xmm10,%xmm15
        movdqa  %xmm5,%xmm9
        psrlq   $4,%xmm5
        movdqa  %xmm3,%xmm10
        psrlq   $4,%xmm3
        pxor    %xmm4,%xmm5
        pxor    %xmm1,%xmm3
        pand    %xmm7,%xmm5
        pand    %xmm7,%xmm3
        pxor    %xmm5,%xmm4
        psllq   $4,%xmm5
        pxor    %xmm3,%xmm1
        psllq   $4,%xmm3
        pxor    %xmm9,%xmm5
        pxor    %xmm10,%xmm3
        movdqa  %xmm0,%xmm9
        psrlq   $4,%xmm0
        movdqa  %xmm15,%xmm10
        psrlq   $4,%xmm15
        pxor    %xmm6,%xmm0
        pxor    %xmm2,%xmm15
        pand    %xmm7,%xmm0
        pand    %xmm7,%xmm15
        pxor    %xmm0,%xmm6
        psllq   $4,%xmm0
        pxor    %xmm15,%xmm2
        psllq   $4,%xmm15
        pxor    %xmm9,%xmm0
        pxor    %xmm10,%xmm15
        movdqa  (%rax),%xmm7
        pxor    %xmm7,%xmm3
        pxor    %xmm7,%xmm5
        pxor    %xmm7,%xmm2
        pxor    %xmm7,%xmm6
        pxor    %xmm7,%xmm1
        pxor    %xmm7,%xmm4
        pxor    %xmm7,%xmm15
        pxor    %xmm7,%xmm0
        .byte   0xf3,0xc3
.size   _bsaes_face_encrypt8,.-
    _bsaes_face_encrypt8

.type   _bsaes_const,@object
.align  64
_bsaes_const:
.MYFIX1:
.quad   0x0000ff0000ff0000, 0
    xff000000000000ff

...
```

A.2 Round Transformation Code of AES-NI-based FACE

A.2.1 Code for 1 x 1

```
static inline void FACE_AESNI_Enc_Block(__m128i *block,
                                        word32 *subkeys,
                                        unsigned int rounds)
{
    unsigned int i;
    const __m128i* skeys = (const __m128i*)(subkeys);

    unsigned char *loc = ((unsigned char *)block) + 15;
```

```
    if(!(*loc))
    {
        *block = _mm_xor_si128(*block, skeys[0]);
        *block = _mm_aesenc_si128(*block, skeys[1]);
        *block = _mm_aesenc_si128(*block, skeys[2]);

        rd2 = _mm_xor_si128 (*block, rd2p_cache[0]);
        rd2p_cache_ptr = rd2p_cache + 1;
    }
    else
    {
        *block = _mm_xor_si128 (*rd2p_cache_ptr, rd2);
        if(++rd2p_cache_ptr == rd2p_cache_ptr_end)
        {
            rd2p_cache_ptr = rd2p_cache + 1;
        }
    }

    *block = _mm_aesenc_si128(*block, skeys[3]);
    *block = _mm_aesenc_si128(*block, skeys[4]);
    *block = _mm_aesenc_si128(*block, skeys[5]);
    *block = _mm_aesenc_si128(*block, skeys[6]);
    *block = _mm_aesenc_si128(*block, skeys[7]);
    *block = _mm_aesenc_si128(*block, skeys[8]);
    *block = _mm_aesenc_si128(*block, skeys[9]);

    if(rounds > 10)
    {
        *block = _mm_aesenc_si128(*block, skeys[10]);
        *block = _mm_aesenc_si128(*block, skeys[11]);
    }

    if(rounds > 12)
    {
        *block = _mm_aesenc_si128(*block, skeys[12]);
        *block = _mm_aesenc_si128(*block, skeys[13]);
    }

    *block = _mm_aesenclast_si128(*block, skeys[rounds]);
}
```

A.2.2 Code for 4 x 1

```
static inline void FACE_AESNI_Enc_4_Blocks(__m128i *block0,
                                           __m128i *block1,
                                           __m128i *block2,
                                           __m128i *block3,
                                           word32 *subkeys,
                                           unsigned int rounds)
{
    unsigned int i;
    const __m128i* skeys = (const __m128i*)(subkeys);
    __m128i rk = skeys[0];

    unsigned char *loc = ((unsigned char *)block0) + 15;

    if(!(*loc))
    {
        *block0 = _mm_xor_si128(*block0, rk);
        *block1 = _mm_xor_si128(*block1, rk);
        *block2 = _mm_xor_si128(*block2, rk);
        *block3 = _mm_xor_si128(*block3, rk);

        rk = skeys[1];
        *block0 = _mm_aesenc_si128(*block0, rk);
        *block1 = _mm_aesenc_si128(*block1, rk);
        *block2 = _mm_aesenc_si128(*block2, rk);
        *block3 = _mm_aesenc_si128(*block3, rk);

        rk = skeys[2];
        *block0 = _mm_aesenc_si128(*block0, rk);
        *block1 = _mm_aesenc_si128(*block1, rk);
        *block2 = _mm_aesenc_si128(*block2, rk);
        *block3 = _mm_aesenc_si128(*block3, rk);
```

```
        rd2 = _mm_xor_si128 (*block0, rd2p_cache[0]);
        rd2p_cache_ptr = rd2p_cache + 4;
    }
    else
    {
        *block0 = _mm_xor_si128 (*rd2p_cache_ptr++, rd2);
        *block1 = _mm_xor_si128 (*rd2p_cache_ptr++, rd2);
        *block2 = _mm_xor_si128 (*rd2p_cache_ptr++, rd2);
        *block3 = _mm_xor_si128 (*rd2p_cache_ptr++, rd2);

        if(rd2p_cache_ptr == rd2p_cache_ptr_end)
        {
            rd2p_cache_ptr = rd2p_cache + 4;
        }
    }

    for (i=3; i<rounds; i++)
    {
        rk = skeys[i];
        *block0 = _mm_aesenc_si128(*block0, rk);
        *block1 = _mm_aesenc_si128(*block1, rk);
        *block2 = _mm_aesenc_si128(*block2, rk);
        *block3 = _mm_aesenc_si128(*block3, rk);
    }

    rk = skeys[rounds];
    *block0 = _mm_aesenclast_si128(*block0, rk);
    *block1 = _mm_aesenclast_si128(*block1, rk);
    *block2 = _mm_aesenclast_si128(*block2, rk);
    *block3 = _mm_aesenclast_si128(*block3, rk);
}
```