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
**Introduction**

- **AES (FIPS 197) algorithm and Counter mode**
  - Used for numerous services as encryption technique
    - OMA DRM v2.0 : PDCF format
    - IPTV, VoIP : SecureRTP
    - SSH, SSL/TLS, and etc.
  - Parallel processing
  - Does not need to implement decryption algorithm
  - Be used in Authenticated Encryption (e.g. GCM, CCM)

![Diagram of AES Algorithm and Structure of Round]

- The counter is incremented for each block.
Introduction

- The feature of AES, which is used in Counter mode
  - 16 bytes counter is increased by 1 (more precisely, pre-defined value) for every block
  - 15 bytes of the counter remain constant for 256 blocks

The counter is incremented for each block.
The feature of AES, which is used in Counter mode

- 16 bytes counter is increased by 1 (more precisely, pre-defined value) for every block
- 15 bytes of the counter remain constant for 256 blocks

< Initial Whitening phase of AES >
Introduction

- The feature of AES, which is used in Counter mode
  - 16 bytes counter is increased by 1 (more precisely, pre-defined value) for every block
  - 15 bytes of the counter remain constant for 256 blocks

- The bottom line
  - Only small changes in each of input-block
  - At the first two rounds, AES operates with many overlapped data
  - These overlapped values can be cached and reused

⇒ Counter-mode Caching**

** HongJun Wu, “Hongjun’s optimized C-code for AES-128 and AES-256”, eSTREAM Project, 2007
Round Function - 4 Transformations

- **SubBytes**
  - Substitutes one byte with another byte (S-Box)
  - Each byte has no relationship with the other
  - Same input always produces same output

- **ShiftRows**
  - Circularly transposes rows from right to left
  - The amount of moving is relevant to the row pos.

- **MixColumns**
  - Is performed column-by-column
  - Combines the four bytes in each column

- **AddRoundKey**
  - Simply XORs a given State with round keys
AES Implementation Methods

- Table-based Implementation
  - Uses pre-computation tables

```c
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xffff2f0dU, 0xd66b6bbdU, 0xde6f6f1bU, 0x91c5c554U,
    0x68303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d76d76U, 0x4dababe6U, 0x76769a9U,
    0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,
    0x7b0b0cbU, 0xa85454fcU, 0x6dbb6bd6U, 0x2c16163aU,
};

static const u32 Te3[256] = {
    0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
    0xf2f20dfeU, 0x6b6b6bb6U, 0x6f6f1bdeU, 0xc5c55491U,
    0x30305060U, 0x01010302U, 0x6767a9ceU, 0x2b2b7d56U,
    0xfefe19e7U, 0xd76d76b5U, 0x0a0b6646U, 0x76769a9eCU,
    0x4141c382U, 0x999b0929U, 0x2d2d775aU, 0x0f0f111eU,
    0xb0b0cb7bU, 0x5454fca8U, 0xb0b0cb7bU, 0x16163a2cU,
};
```

```c
s0 = GETU32(in     ) ^ rk[0];
s1 = GETU32(in +  4) ^ rk[1];
s2 = GETU32(in +  8) ^ rk[2];
s3 = GETU32(in + 12) ^ rk[3];

/* round 1: */
t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
```
AES Implementation Methods

- **Table-based Implementation**
- Uses pre-computation tables

```c
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bddU, 0xde6f6fb1U, 0x91c5c554U,
    0x6f303050U, 0x02010103U, 0x8e6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xc76769aU,
    ... 0x4141c382U, 0x9999b029U, 0x2d2d775aU, 0x0f0f111eU,
    0xb0b0cb7bU, 0x5454fca8U, 0xbbbd66d6U, 0x16163a2cU,
};

static const u32 Te3[256] = {
    0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
    0xff2f20ffU, 0xb66b6bddU, 0xf66f6fb1U, 0xc5c55491U,
    0x30305060U, 0x01010302U, 0x6767a9ceU, 0x2b2b7d56U,
    0xfefe19e7U, 0xd7d762b5U, 0x4dababe6dU, 0x76769a8fU,
    ... 0x4141c382U, 0x9999b029U, 0x2d2d775aU, 0x0f0f111eU,
    0xb0b0cb7bU, 0x5454fca8U, 0xbbbd66d6U, 0x16163a2cU,
};
```

Vulnerable to **Cache timing attack**

```c
s0 = GETU32(in     ) ^ rk[0];  
s1 = GETU32(in +  4) ^ rk[1];  
s2 = GETU32(in +  8) ^ rk[2];  
s3 = GETU32(in + 12) ^ rk[3];

/* round 1: */
t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
```
**AES Implementation Methods**

**Bitsliced Implementation**
- First proposed by Biham to improve the software performance of DES (1997)
- Simulates a hardware implementation in software (sequence of Boolean operations)

< 8 plaintext blocks >

<table>
<thead>
<tr>
<th>Block</th>
<th>MSB</th>
<th>...</th>
<th>LSB</th>
<th>8 [128-bits] registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block0:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000000</td>
</tr>
<tr>
<td>Block1:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000001</td>
</tr>
<tr>
<td>Block2:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000010</td>
</tr>
<tr>
<td>Block3:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000011</td>
</tr>
<tr>
<td>Block4:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000100</td>
</tr>
<tr>
<td>Block5:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000101</td>
</tr>
<tr>
<td>Block6:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000110</td>
</tr>
<tr>
<td>Block7:</td>
<td>b0 b1 b2 b3</td>
<td>...</td>
<td>b12 b13 b14</td>
<td>00000111</td>
</tr>
</tbody>
</table>

< bitsliced form transformation (OpenSSL implementation based on [1]) >

AES Implementation Methods

**AES-NI (Hardware acceleration)**

- Intel supports AES instruction set since Westmere processor (in March 2008)
- Support 7 instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESENC</td>
<td>Perform one round of an AES encryption flow</td>
</tr>
<tr>
<td>AESENCLAST</td>
<td>Perform the last round of an AES encryption flow</td>
</tr>
<tr>
<td>AESDEC</td>
<td>Perform one round of an AES decryption flow</td>
</tr>
<tr>
<td>AESDECLAST</td>
<td>Perform the last round of an AES decryption flow</td>
</tr>
<tr>
<td>AESKEYGENASSIST</td>
<td>Assist in AES round key generation</td>
</tr>
<tr>
<td>AESIMC</td>
<td>Assist in AES Inverse Mix Columns</td>
</tr>
<tr>
<td>PCLMULQDQ</td>
<td>Carryless multiply</td>
</tr>
</tbody>
</table>

*block = _mm_xor_si128( *block , skeys[0] ) ;

/* round 1: */
*block = _mm_aesenc_si128 ( *block , skeys[1] ) ;
# AES Implementation Methods

## Fastest throughput of each method

<table>
<thead>
<tr>
<th>Method</th>
<th>Performance (Cycles per Byte)</th>
<th>Test Environment</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table-based</td>
<td>$10.57 + \alpha$ (not for CTR)</td>
<td>Core 2 Quad Q6600</td>
<td>INDOCRYPT 2008 [1]</td>
</tr>
<tr>
<td>Bitslicing</td>
<td>9.32</td>
<td>Core 2 Quad Q6600</td>
<td>CHES 2009 [2]</td>
</tr>
<tr>
<td></td>
<td><strong>7.59</strong></td>
<td>Core 2 Quad Q9550</td>
<td></td>
</tr>
<tr>
<td>AES-NI</td>
<td>1.4 – 2.0</td>
<td>Westmere Processor</td>
<td>INTEL whitepaper [3]</td>
</tr>
<tr>
<td></td>
<td><strong>0.57</strong></td>
<td>Skylake Core i5</td>
<td>Crypto++ Benchmark [4]</td>
</tr>
</tbody>
</table>

( The first Westmere-based processors (that supports AES-NI) were launched on Jan, 2010. )
Problem

- Previous Counter-mode Caching can not work effectively on bitsliced and AES-NI-based implementations
  - It only covered partial data of round transformation
    → The rest (which was not cached) should be calculated in every block

During a format conversion, each byte of input is sliced bitwise. And the sliced bits are spread in the corresponding positions of each register.

Necessary input bytes to calculate the rest are spread to whole register

Almost the whole instructions of previous implementation should be performed with additional operations (save, load, merge)

Adding some operations to calculate the rest becomes a considerable burden even if instruction latency and throughput differ from each instruction

Such operations (for the rest) should be composed of several instructions

aesenc xmm15, xmm1 → only 1 instruction performs round operation
Our Work (FACE)

- **We propose an efficient implementation technique for the CTR mode of AES (FACE)**
  - Extends the counter-mode caching
  - Can be employed, regardless of the platform, environment, or implementation method

- **We show that FACE can be applied to existing implementation methods**
  - Table-based, bitsliced, and AES-NI-based implementations
  - The first to combine counter-mode caching with bitsliced implementation
  - The first to apply counter-mode caching up to the round transformations of AES-NI implementation

- **Our proposal (FACE) records the highest throughput ever achieved**
  - Bitslice: 6.41 cycles/byte on an Intel Core 2 Q9550 (previous record: 7.59 cycles/byte)
  - AES-NI: 0.44 cycles/byte on an Intel Core i7 8700K (previous record: 0.55 cycles/byte)
FACE (Fast AES Counter Mode Encryption)

5 types of reuse techniques

- **FACE_{rd0}**
  - Cache 12 bytes of round 0’s result
  - Reuse for $2^{32} - 1$ successive blocks

- **FACE_{rd1}**
  - Cache 12 bytes of round 1’s result
  - Reuse for 255 successive blocks

- **FACE_{rd1+}**
  - Generate Pre-computation Table (1K)
  - Reuse for $2^{40}$ successive blocks

- **FACE_{rd2}**
  - Cache 16 bytes of round 2
  - Reuse for 255 successive blocks

- **FACE_{rd2+}**
  - Generate Pre-computation Table (4K)
  - Reuse for $2^{40}$ successive blocks
Fast AES Counter mode Encryption

$\text{FACE}_{\text{rd0}}$

- 16 bytes counter is increased by 1 for every block
- 15 bytes of the counter remain constant for 256 block
- The difference between one block and next block is just last 1 byte
Fast AES Counter mode Encryption

**FACE**

**FACE**

- 16 bytes counter is increased by 1 for every block
- 15 bytes of the counter remain constant for 256 block
- The difference between one block and next block is just last 1 byte
Fast AES Counter mode Encryption

16 bytes counter is increased by 1 for every block

15 bytes of the counter remain constant for 256 block

The difference between one block and next block is just last 1 byte

Cache 3 columns of Initial whitening (Round 0)

The cached value can be reused in $2^{32} - 1$ consecutive blocks
The difference between two input blocks is just last 1 byte.
Fast AES Counter mode Encryption

- **The difference between two input blocks is just last 1 byte**
- **This difference spreads by ShiftRows() and MixColumns() operation**
Fast AES Counter mode Encryption

- **The difference** between two input blocks is just **last 1 byte**
- **This difference spreads** by ShiftRows() and MixColumns() operation
- **Cache 3 columns** of Round 1 result (12 bytes)
- The cached value can be reused in **255 consecutive blocks**
Fast AES Counter mode Encryption

- Generate **Pre-computation lookup table** (size: 1KB)

![Diagram of AES encryption process with tables and annotations]

<table>
<thead>
<tr>
<th>Round 0</th>
<th>Round 1</th>
<th>Round 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubBytes</td>
<td>ShiftRows</td>
<td>AddRoundKey</td>
</tr>
<tr>
<td>MixColumns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**1st Block:**

|------|------|------|------|

**2nd Block:**

|------|------|------|------|

- $ctr[15] \oplus round_0's rk[15]$
Fast AES Counter mode Encryption

**FACE**

- **Generate** Pre-computation lookup table (size: 1KB)
- **Store and Reuse** the first column of round 1
- The lookup table can be used in $2^{40}$ consecutive blocks (1,099,511,627,776 block = 17,592,186,044,416 bytes = 16 TB)
- The lookup index is the last byte of the counter
Fast AES Counter mode Encryption

Leverage $\text{FACE}_{rd1}$ & $\text{FACE}_{rd1+}$

**Counter**
- $b_0$, $b_1$, $b_2$, ..., $b_{14}$, $b_{15}$

**Initial whitening (Round 0)**

**Caching Procedure (Round 1)**

- **Index** ($b_{15}$) | **State Value (4 Bytes)**
  - 0  | 0x64E2F9C1
  - 1  | 0x1A83B211
  - 254  | 0x73816F1F
  - 255  | 0x6C8EB21D

**Caching Procedure (Round 1)**

- **S[0]** | **S[1]** | **S[2]** | **S[3]**
- **S[4]** | **S[5]** | **S[6]** | **S[7]**
- **S[8]** | **S[9]** | **S[10]** | **S[11]**
- **S[12]** | **S[13]** | **S[14]** | **S[15]**

- **SubBytes**
- **ShiftRows**
- **MixColumns**
- **AddRoundKey**

**Round 1**
- Complete (Up to Round 1) after memory load and merge

**Round 2**

- **Look-up Table**

**Saved State ($\text{FACE}_{rd1}$)**
The difference between two input blocks (into r2) is the first column (4 bytes).
Fast AES Counter mode Encryption

- The difference between two input blocks (into r2) is the first column (4 bytes)
- This difference spreads to all States by ShiftRows() and Mixcolumns() operation
The difference between two input blocks (into r2) is the first column (4 bytes).

This difference spreads to all States by ShiftRows() and Mixcolumns() operation.

Cache intermediate result of MixColumn() operation (16 bytes).

The cached value can be reused in 255 consecutive blocks.
Fast AES Counter mode Encryption

- **Pre-computation lookup table** (size: 4KB)

1. **1st Block:**
   - Round 1:
     - S0 | S1 | S2 | S3
     - S4 | S5 | S6 | S7
     - S8 | S9 | S10 | S11
   - Round 2:
     - S0 | S1 | S2 | S3
     - S4 | S5 | S6 | S7
     - S8 | S9 | S10 | S11
   - Round 3:
     - S0 | S1 | S2 | S3
     - S4 | S5 | S6 | S7
     - S8 | S9 | S10 | S11

2. **2nd Block:**
   - Round 1:
     - S0 | S1 | S2 | S3
     - S4 | S5 | S6 | S7
     - S8 | S9 | S10 | S11
   - Round 2:
     - S0 | S1 | S2 | S3
     - S4 | S5 | S6 | S7
     - S8 | S9 | S10 | S11
   - Round 3:
     - S0 | S1 | S2 | S3
     - S4 | S5 | S6 | S7
     - S8 | S9 | S10 | S11
**FACE**

**FACE_{rd2+}

- **Generate Pre-computation lookup table** (size : 4KB)
- **Store and Reuse** intermediate result of MixColumns() operation

By FACE_{rd1+}, \( \text{ctr}[15] \) determines

---

**Fast AES Counter mode Encryption**

**1ST Block:**

**2nd Block:**

**Round 1**

**Round 2**

**Round 3**

**SubBytes**

**ShiftRows**

**AddRoundKey**

**MixColumns**

**MixColumns**

**The case of State[0]**

\[
\begin{bmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{bmatrix}
\]

- Round Key:

\[
\begin{bmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{bmatrix}
\]

**Round Key:**
Fast AES Counter mode Encryption

- **Pre-computation lookup table** (size: 4KB)
- **Store and Reuse** intermediate result of MixColumns() operation
- The lookup table can be used in $2^{40}$ consecutive blocks
  (1,099,511,627,776 block = 17,592,186,044,16 bytes = 16 TB)
- The lookup index is the last byte of the counter

![Diagram of AES Counter mode](https://example.com/diagram.png)
Fast AES Counter mode Encryption

**FACE**

**FACE rd2**

- Generate **Pre-computation lookup table** (size: 4KB)
- Store and Reuse intermediate result of `MixColumns()` operation
- The lookup table can be used in $2^{40}$ consecutive blocks
  (1,099,511,627,776 block = 17,592,186,044,416 bytes = 16 TB)

**The whole operations up to round 2 can be done by 2 memory load and 1 XOR operations only!**
Cache timing Attacks

- Exploits timing differences between accessing *cached* vs. *non-cached* data
  - CacheBleed: cache-bank conflicts

- Software countermeasures
  - Constant-time implementation
    → ensures that secret information is not disclosed through the operation of the code
  - To be constant time
    - only uses *fixed-time instructions* with arguments that depend on secret data
    - does not use conditional branches that depend on secret data
    - does not use memory access patterns that depend on secret data
**Cache timing Attacks**

- Exploits timing differences between accessing **cached** vs. **non-cached** data
  - **CacheBleed**: cache-bank conflicts

- **Software countermeasures**
  - Constant-time implementation
    → ensures that secret information is not disclosed through the operation of the code
  - To be constant time
    - only uses **fixed-time instructions** with arguments that **depend on secret data**
    - does not use **conditional branches** that **depend on secret data**
    - does not use **memory access patterns** that **depend on secret data**

- Our method looks like vulnerable to timing attacks (the use of lookup tables)

- But, **FACE** has no operations that depend on secret data
  - In case of **FACE_{rd0}**, **FACE_{rd1}**, and **FACE_{rd2}**, the size of cache is small and the indices are fixed (i.e. constant data)
  - In case of **FACE_{rd1+}** and **FACE_{rd2+}**, the index is merely a part of counter that does not need to be secret and the index increases linearly
Evaluations

Implementation

- We implement FACE by modifying the AES source code contained in the open-source libraries
  - we select targets which can be considered as the fastest one
  - OpenSSL : table-based and bitsliced
    - [BS08]* is the fastest table-based implementation. But table-based is not our main targets.
    - Bitsliced AES is implemented based on [KS09]** (the fastest bitsliced implementation)
  - Crypto++ : AES-NI
    - Throughput records based on eSTREAM/Crypto++ benchmark, and etc

- For a fair comparison, we did not re-code the existing strategy into our own implementation (the quality of code)
  \[ \text{Except for our strategy, all other conditions remain the same} \]

<table>
<thead>
<tr>
<th></th>
<th>Test Env_1</th>
<th>Test Env_2</th>
<th>Test Env_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core 2 Quad Q9550</td>
<td>Intel Core i7 4770K</td>
<td>Intel Core i7 8700K</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.8 GHz</td>
<td>3.5 GHz</td>
<td>3.7 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
<td>8 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 3.19.0-32 x86_64</td>
<td>Linux 3.19.0-32 x86_64</td>
<td>Linux 4.13.0-36 x86_64</td>
</tr>
</tbody>
</table>

* [BS08]: Daniel J Bernstein and Peter Schwabe, “New AES software speed records”, INDOCRYPT 2008
** [KS09]: Emilia Käsper and Peter Schwabe, “Faster and timing-attack resistant AES-GCM”, CHES 2009
## Evaluations

### Experimental results (Throughput)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Implementation</th>
<th>Method</th>
<th>Target</th>
<th>1024 bytes</th>
<th>4096 bytes</th>
<th>20480 bytes</th>
<th>40960 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>This Paper</td>
<td>Table-based</td>
<td>8.380</td>
<td>10.085</td>
<td>11.808</td>
<td>8.344</td>
<td>10.064</td>
</tr>
<tr>
<td></td>
<td>This Paper (R1)</td>
<td>AES-NI</td>
<td>1.025</td>
<td>1.287</td>
<td>1.556</td>
<td>1.018</td>
<td>1.253</td>
</tr>
<tr>
<td></td>
<td>This Paper (R2)</td>
<td>AES-NI</td>
<td>0.927</td>
<td>1.160</td>
<td>1.383</td>
<td>0.917</td>
<td>1.146</td>
</tr>
<tr>
<td>4 x 1</td>
<td>Crypto++</td>
<td>AES-NI</td>
<td>0.730</td>
<td>0.861</td>
<td>0.984</td>
<td>0.704</td>
<td>0.840</td>
</tr>
<tr>
<td></td>
<td>This Paper (R1)</td>
<td>AES-NI</td>
<td>0.634</td>
<td>0.781</td>
<td>0.923</td>
<td>0.623</td>
<td>0.769</td>
</tr>
<tr>
<td></td>
<td>This Paper (R2)</td>
<td>AES-NI</td>
<td>0.592</td>
<td>0.727</td>
<td>0.869</td>
<td>0.580</td>
<td>0.714</td>
</tr>
<tr>
<td>1 x 1</td>
<td>Crypto++</td>
<td>AES-NI</td>
<td>1.865</td>
<td>1.871</td>
<td>2.059</td>
<td>1.625</td>
<td>1.847</td>
</tr>
<tr>
<td></td>
<td>This Paper (R1)</td>
<td>AES-NI</td>
<td>0.778</td>
<td>0.867</td>
<td>0.986</td>
<td>0.739</td>
<td>0.827</td>
</tr>
<tr>
<td></td>
<td>This Paper (R2)</td>
<td>AES-NI</td>
<td>0.703</td>
<td>0.786</td>
<td>0.880</td>
<td>0.662</td>
<td>0.775</td>
</tr>
<tr>
<td>4 x 1</td>
<td>Crypto++</td>
<td>AES-NI</td>
<td>0.551</td>
<td>0.669</td>
<td>0.767</td>
<td>0.547</td>
<td>0.642</td>
</tr>
<tr>
<td></td>
<td>This Paper (R1)</td>
<td>AES-NI</td>
<td>0.513</td>
<td>0.607</td>
<td>0.706</td>
<td>0.494</td>
<td>0.586</td>
</tr>
<tr>
<td></td>
<td>This Paper (R2)</td>
<td>AES-NI</td>
<td>0.450</td>
<td>0.547</td>
<td>0.638</td>
<td>0.441</td>
<td>0.533</td>
</tr>
</tbody>
</table>
Conclusion

- AES-CTR is used for numerous high throughput applications

- We propose FACE, which can improve the performance of AES CTR by using repetitive data (approximately 15-20% improved)

- FACE can be employed in any AES CTR implementation, regardless of implementation method (i.e. table-based, bitsliced, and AES-Ni-based)

- And further... Verify whether caching strategy can be applied to other algorithms that have similar characteristics to the AES CTR (e.g. CAESAR finalist Deoxys)
Conclusion

- AES-CTR is used for numerous high throughput applications
- We propose FACE, which can improve the performance of AES CTR by using repetitive data (approximately 15-20% improved)
- FACE can be employed in any AES CTR implementation, regardless of implementation method (i.e. table-based, bitsliced, and AES-NI-based)
- And further... Verify whether caching strategy can be applied to other algorithms that have similar characteristics to the AES CTR (e.g. CAESAR finalist Deoxys)

Thank you for your attention!
Any Questions?