An Efficient Indexing Technique for AES Lookup Table to prevent Side-Channel Cache-Timing Attack

Md. Refazul Islam Refat[a], Md. Mosaddek Khan*[a], Euna Islam[a], Md. Mamum-or-Rashid[a]

[a]*Department of Computer Science and Engineering, University of Dhaka, Bangladesh*

Abstract

In the era of virtualization, co-residency with unknown neighbours is a necessary evil and leakage of information through side-channels is an inevitable fact. Preventing this leakage of information through side-channels, yet maintaining high efficiency, has become one of the most challenging parts of any implementation of the Advanced Encryption Standard (AES) based on the Rijndael Cipher. Exploiting the associative nature of the cache and susceptible memory access pattern, AES is proved to be vulnerable to side-channel cache-timing attacks. The reason of this vulnerability is primarily ascribed to the existence of correlation between the Bytes of the state matrix and corresponding accessed memory blocks. In this paper, we idealized the model of cache-timing attack and proposed a way of breaking this correlation through the implementation of a Random Address Translator (RAT). The simplicity of the design architecture of RAT can make itself a wonderful choice as a way of indexing the lookup tables for the implementers of the AES seeking resistance against side-channel cache-timing attacks.

Keywords: AES, Cache-Timing Attack, Security

1. Introduction

Out of the five candidate algorithms (MARS [1], RC6, Rijndael [2], Serpent [3], and Twofish [4]) for the AES, Rijndael was pronounced as a new standard on November 26, 2001 as FIPS PUB 197 [5]. The security barrier of Rijndael cipher is so strong that a cryptographic break is infeasible with current technology. Even Bruce Schneier, a developer of the competing algorithm Twofish, admired Rijndael cipher in his writing, "I do not believe that anyone will ever discover an attack that will allow someone to read Rijndael traffic" [6].

A brute force method would require 2^{128} operations for the full recovery of an AES-128 key. A machine that can perform 8.2 quadrillion calculations per second will take 1.3 quadrillion years to recover this key. However, partial information leaked by side-channels can tear down down this complexity to a very reasonable level. Side-channel attacks do not attack the underlying cipher; they rather attack implementations of the cipher on systems that inadvertently leak data. Ongoing research in the last decade has shown that the information transmitted via side-channels, such as execution time [7], computational faults [8], power consumption [9] and electromagnetic emissions [10, 11, 12], can be detrimental to the security of Rijndael [13] and other popular ciphers like RSA [14].

We will be primarily focusing on the vulnerability of Rijndael AES to side-channel cache-timing attack and see how simple cache misses can lead to dire consequences. The prevention is not trivial; there exists a trade-off between performance and degree of multiprogramming. We will present a concept that might balance this trade-off by introducing a little memory overhead. The biggest advantage could be that even

Email addresses: refazul.refat@gmail.com (Md. Refazul Islam Refat), mosaddek@cse.univdhaka.edu (Md. Mosaddek Khan*), euna.islam@gmail.com (Euna Islam), mamun@cse.univdhaka.edu (Md. Mamum-or-Rashid)

though the process is arbitrary, the cipher text will always remain the same.

2. Related Works

In October 2005, Dag Arne Osvik, Adi Shamir and Eran Tromer presented a paper demonstrating several cache-timing attacks against AES [15]. One attack was able to obtain an entire AES key after only 800 triggering encryptions, in a total of 65 milliseconds. The attack required the attacker to be able to run programs on the same system performing AES. That paper presented some decent defence against side-channel cache-timing attacks such as avoiding memory access, disabling cache sharing, dynamic table storage etc.

They revised their paper on 2010 [16] and focused on modern microprocessor implementations and presented few more preventions. Most of them lacked conceptual consolidation and provided as implicit implication only. Some of them even requires alteration of cipher text that might cause significant overhead in distributed systems. Other works included introduction of asynchronous circuitry in AES implementation [7].

3. Preliminaries

Before understanding the weakest part of the cipher, a clarified concept of cache memory and how they are accessed during real time AES implementation might be helpful.

The memory access patterns of AES are particularly susceptible to cryptanalysis. The cipher is abstractly defined by algebraic operations and could, in principle, be directly implemented using just logical and arithmetic operations. However, performance-oriented software implementations on 32-bit (or higher) processors typically use an alternative formulation based on lookup tables as prescribed in the Rijndael Specification [13].

Several lookup tables are precomputed once by the programmer or during system initialization. There are 8 such tables, T_0, T_1, T_2, T_3 and $T_0^{10}, T_1^{10}, T_2^{10}, T_3^{10}$, each containing 256 4-byte words. The contents of the tables, defined in [13], are inconsequential to most of the cache-timing attacks because of memory protection.

During key setup, a given 16-byte secret key $k = (k_0, k_1, ..., k_{15})$ is expanded into 10 round keys, $K^{(r)}$ for $r = 1, 2, ..., 10$. Each round is divided into 4 words of 4 bytes each: $K^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$. The 0th round key is just the raw key: $K_j^{(0)} = (k_{4j}, k_{4j+1}, k_{4j+2}, k_{4j+3})$ for $j = 0, 1, 2, 3$. The details of the rest of the key expansions are mostly inconsequential [16].

Given a 16-byte plaintext $p = (p_0, p_1, ..., p_{15})$, encryption proceeds by computing a 16-byte intermediate state matrix $x^{(r)} = (x_0^{(r)}, x_1^{(r)}, ..., x_{15}^{(r)})$ at each round r. The inital state matrix $x^{(0)}$ is computed by $x_i^{(0)} = p_i \oplus k_i (i = 0, 1, ..., 15)$.

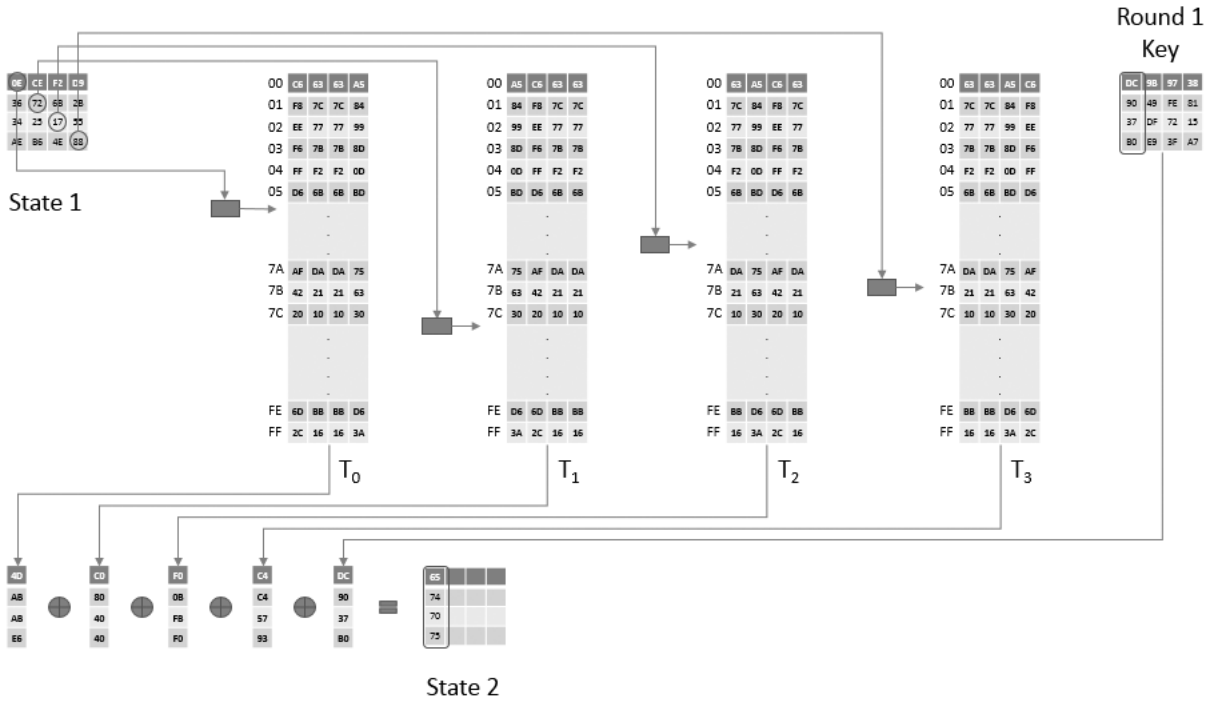

Figure 1: The figure demonstrates how $x_0^1, x_1^1, x_2^1, x_3^1$ are computed. Note that the bytes of the state matrix are being used as index of the lookup tables. Since the lookup table remains the same for a particular S-Box, an index byte will always cause lookup from the same block of cache memory

The first 9 rounds are computed by updating the intermediate state as follows, for $r = 0, 1, .., 8$:

$$(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) \leftarrow T_0[x_0(r)] \oplus T_1[x_5(r)] \oplus T_2[x_{10}(r)] \oplus T_3[x_{15}(r)] \oplus K_0^{(r+1)}$$
$$(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) \leftarrow T_0[x_4(r)] \oplus T_1[x_9(r)] \oplus T_2[x_{14}(r)] \oplus T_3[x_3(r)] \oplus K_0^{(r+1)}$$
$$(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) \leftarrow T_0[x_8(r)] \oplus T_1[x_{13}(r)] \oplus T_2[x_2(r)] \oplus T_3[x_7(r)] \oplus K_0^{(r+1)}$$
$$(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}) \leftarrow T_0[x_{12}(r)] \oplus T_1[x_1(r)] \oplus T_2[x_6(r)] \oplus T_3[x_{11}(r)] \oplus K_0^{(r+1)}$$

Finally, to compute the last round, the above process of table-lookup is repeated with $r = 9$, except that T_0, T_1, T_2, T_3 are replaced by $T_0^{(10)}, T_1^{(10)}, T_2^{(10)}, T_3^{(10)}$. The resulting $x^{(10)}$ is the ciphertext. Compared to the algebraic formulations of AES, the Lookup tables here represent the combination of *ShiftRows*, *MixColumns* and *SubBytes* operations; the change of lookup tables in the last round is due to the absence of *MixColumns* operation. It is clear from the above discussion that computation of a state matrix requires 16 Table lookups and 16 XOR operations.

4. The Idealized Prime+Probe Technique

The idealized Prime+Probe technique helps to visualize the vulnerability of AES cipher during the 1st Round. To keep things simple, we'll consider an idealized environment in which only the attacking process and the victim encrypting process exists. Also assume that the attacking process will be able to invoke or interrupt the encrypting process at any time. Now since we are talking about triggered encryption, the plaintext is always known.

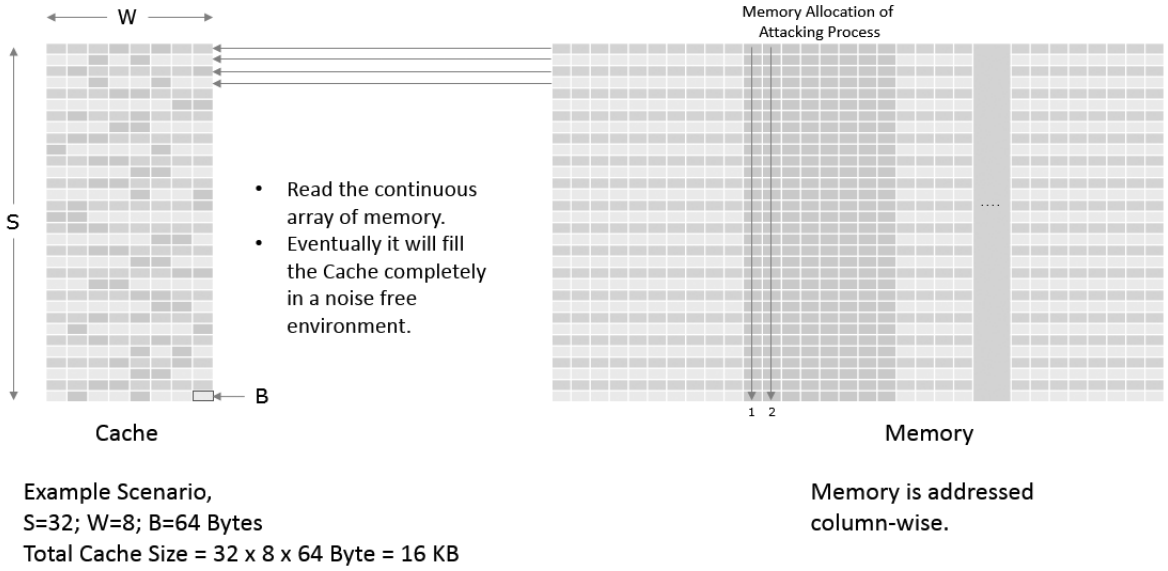

Figure 2: Demonstration of the Cache filling process. Note that reading continuous array of memory will cause allocation of one block from each cache set.

The Prime+Probe technique has the following 3 phases:

- Prime
- Trigger
- Probe

Prime. During the Prime phase, the cache is filled completely by the attacking process. This can be done by allocating a contiguous amount of memory as big as the size of the cache and performing a read on it.

Trigger. With the cache filled with attacker's data, he will trigger an encryption for some known plain text. As the encrypting process reaches the first round and performs table lookup, it is interrupted. An important point to note here is, the model is being idealized to illustrate the susceptibility of the first round. Recovering the AES Encryption Key is not the intent of this paper.

Probe. After the encrypting process is interrupted just after it performs its table lookup during the first round, the Prime step is done again, i.e. the cache is filled again. But this time, a clock is maintained to determine a cache miss. Before the Trigger phase, the cache was completely filled by attacker's data. But after that Trigger phase, one block of the Lookup table T_i is now in the cache due to lookup operation. So all but that block will suffer a cache miss.

Observing the position of the cache miss, attacker can easily identify which block of T_i was accessed since it was assumed that Lookup Tables T_i are congruent with the cache. Since size of *memory block* is 64 Bytes in most cases, each *memory block* holds 16 entry of the Lookup Table. So for example, if 7th block of T_0 was accessed, one can conclude the index $x_0^{(1)}$ was somewhat between 0x60 and 0x6F, i.e. the high nibble is 0x6. We know

$$x_0^{(1)} = x_0^{(0)} \oplus k_0^{(0)}$$

Algorithm 1 Abstract procedure for capturing half of the Key during the first round

```
triggerEncryption()
for i ← 0 to 15 do
    fillCache()
    haltEncryption()
    j ← missedBlockIndex()
    j ← j ≪ 4
    k_i^(0) ← j ⊕ x_i^(0)        ▷ k_i^(0) and x_i^(0) resemble i-th Byte of the key and known plaintext respectively
end for
```

Here $x_0^{(0)}$ is the 1st Byte of the plaintext and $k_0^{(0)}$ is the 1st Byte of the Encryption Key. Now, since addition and subtraction in Galois Field are the same XOR operation, rearranging the equation yields:

$$k_0^{(0)} = x_0^{(0)} \oplus x_0^{(1)}$$

knowing $x_0^{(0)}$ and the high nibble of $x_0^{(1)}$, one can compute the high nibble of the first Byte of the Encryption Key by a simple XOR operation. Proceeding this way for every Bytes $x_i^{(1)}$ for $i = 0, 1, 2, ..., 15$, one can dig up half of the Encryption Key. The idealized procedure is summarized in algorithm 1.

5. Proposed Random Address Translator (RAT)

From the analysis in the previous sections, it is prominent that the existence of linear correlation between index Bytes of the state matrix and the location of corresponding accessed block in memory is the primary reason of vulnerability of AES in the first round. Given size of memory block (64 Bytes in most of the cases), any information about the memory access during the encryption process would directly reveal high nibble of the index Bytes. Thus if the size of the key is 128 Bit, pre-knowledge of the high nibbles of the index Bytes would reduce the brute force complexity to 2^{64}.

If the correlation can be broken, so that the index Bytes will no longer be used to directly locate the desired block of the Lookup table, most of the cache-timing threats can be eliminated. The key concept to the solution is to perform memory access to get the desired content from an arbitrary, unpredictable location. This can be accomplished by introducing a Translator table between the state matrix and the Lookup table.

The most common picture that one might perceive in such a scenario is the mediation of a mapping table that maps one Byte to another. But a flat mapping table would have several disadvantages. First of all, it will incur more cache misses than a normal table lookup encryption process. This may account for disatrous performance concern because cache misses are very expensive. Secondly, it can be large; at least 256 Bytes since the Lookup table itself has 256 entries. A good translator must consume least number of Bytes as part of its mapping operation as well as seek for at most the same number of cache misses that would occur in a normal table lookup operation.

One reasonable way to ensure similar number of cache misses as the normal process would be to keep chunks of Bytes of the translator table together and then rearrange them. In our solution, we defined 1 chunk to contain 16 Bytes. So we have a total of 16 chunks and if the memory block (Cache Line) is at least 64 Bytes, it can be easily inferred that we are not incurring more cache misses than the normal process. At present, most processors have 64 Byte cache line.

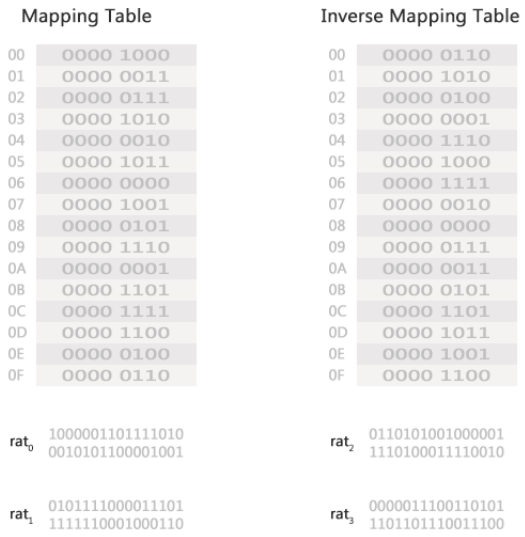

Figure 3: The figure illustrates the state of the mapping and inverse mapping tables after randomization. Since numbers from 0 to 15 requires only 4 bits and a Byte is capable of storing 8 bits, the tables can be condensed by a factor of two and can be conveniently stored as a set of four 32 bit integers rat_i for $0 <= i <= 3$

The next big concern is size. Mapping 256 Bytes requires another 256 Bytes, but mapping 256 Bytes with 16 Bytes grouped together requires only 16 Bytes. Let $f(x)$ be that mapping function. Then f(0x2E) can return anything from the set {0x0E,0x1E,0x2E,...,0xFE} but not something like 0x2F or 0xAB. That is, the low nibble will be unchanged.

Algorithm 2 Constructing the RAT

function INIT()
  $x \leftarrow \{0, 1, 2, ..., 15\}$
  $y \leftarrow \{0, 1, 2, ..., 15\}$
  $rat \leftarrow \{0, 0, 0, 0\}$
  for Arbitrary number of times **do**
    $i \leftarrow rand()$ mod 16 ▷ mod resembles modulus operation
    $j \leftarrow rand()$ mod 16
    $swap(x_i, x_j)$
    $y_{x_j} \leftarrow i$
    $y_{x_i} \leftarrow j$
  end for
  for $i \leftarrow 0$ to 7 **do**
    $rat_0 \leftarrow rat_0 + x_i \ll ((7 - i) \ll 2)$ ▷ $\ll$ resemebles bitwise left shift operation
    $rat_1 \leftarrow rat_1 + x_{i+8} \ll ((7 - i) \ll 2)$
    $rat_2 \leftarrow rat_2 + y_i \ll ((7 - i) \ll 2)$
    $rat_3 \leftarrow rat_3 + y_{i+8} \ll ((7 - i) \ll 2)$
  end for
  delete x,y
  return rat
end function

Algorithm 2 outlines the initialization process. It takes no parameter and returns a set of four 32 bit integers. The first two integers rat_0 and rat_1 will be used randomize the Lookup table by algorithm (3)

and (4). The last two integers rat_2 and rat_3 will be used to retrieve correct piece of information from the Lookup table.

Algorithm 3 Mapping Fuction

```
1: function MAP(i)
2:     j ← 1
3:     if ((i & 128) = 0) then                    ▷ & resembles bitwise and operation
4:         j ← 0
5:     end if
6:     k ← ((255 − i) ≫ 4) ≪ 2    ▷ ≪ and ≫ resemble bitwise left and right shift operations respectively
7:     x ← (rat_j ≫ k) & 15
8:     y ← i & 15
9:     return (x ≪ 4) + y
10: end function
```

To illustrate what this mapping function does, let us assume that map(0x3E) returns 0x9E. This implies that the content of Lookup table at offset 0x9E is now at offset 0x3E. The following $mingle()$ function does the randomization. It takes the original Lookup table as argument and returns the confounded Lookup table. A temporary table is created to catalyze the process. After the randomization is done, that temporary table is removed from memory.

Algorithm 4 Mingling Function

```
1: function MINGLE(table)                ▷ Takes unmodified lookup table as parameter
2:     for i ← 0 to 255 do
3:         for j ← 0 to 3 do
4:             temp_{i,j} ← table_{map(i),j}
5:         end for
6:     end for
7:     for i ← 0 to 255 do
8:         for j ← 0 to 3 do
9:             table_{i,j} ← temp_{i,j}
10:        end for
11:    end for
12:    delete temp
13:    return table
14: end function
```

Now that we have a perplexed Lookup table, encryption can not still be done because the pieces are not in their right place. Here the last two integers rat_2 and rat_3 come into action. Their job is to get the correct piece of information from the mixed up Lookup table. For that we need an inverse mapping function. For example, if map(0x3E) returns 0x9E, $inverseMap$(0x9E) would return 0x3E. Now all that need to be done is: pass Bytes to the inverse mapping function and use the returned Bytes as index to the scrambled Lookup table.

The overall picture is, instead of using Bytes of the state matrix as offset to the Lookup table, the Lookup table itself is scrambled and an address translator is introduced between the state matrix and Lookup table. The job of this translator is to map one Byte to another to get the correct offset for the scrambled Lookup table. If the Lookup is scrambled in group of 64 Bytes, this translator table will require only 8 Bytes that can be conveniently stored in two 32 bit integers.

Algorithm 5 Inverse Mapping Fuction

```
1: function inverseMap(i)
2:     j ← 3
3:     if ((i & 128) = 0) then                    ▷ & resembles bitwise and operation
4:         j ← 2
5:     end if
6:     k ← ((255 − i) ≫ 4) ≪ 2    ▷ ≪ and ≫ resemble bitwise left and right shift operations respectively
7:     x ← (rat_j ≫ k) & 15
8:     y ← i & 15
9:     return (x ≪ 4) + y
10: end function
```

6. Performance Analysis

There are some other proposed solutions in this field [16, 17, 18, 19, 20, 21, 22]. We take some of the most prominent ones and compare it with our proposed solution.

Avoiding Memory Access. Since the cache-timing attacks exploit the effect of memory access on the cache, any implementation that does not perform any table lookup won't suffer from this sort of attacks. But this solution has a major drawback. The performance is degraded by an order of magnitude. We need a total of 24 shift operations, 16 table lookups, 8 conditional XOR operations, 28 mandatory XOR operations for a single round in a straightforward implementation.

In fact, the conditional XOR operation is vulnerable to timing attack too [23]. Other than lagging behind in terms of security, straightforward implementation requires more computation time. But the positive side is, straightforward implementation doesn't need to maintain any sort of lookup table other than the S-Box. On the other hand, table lookup with RAT has an overhead of mapping and inverse mapping during encryption. From algorithm 5 it is prominent that a single call to inverse mapping function incurs 4 shift operations, 3 AND operations, 2 addition operations on a single integer. This has very intangible effect on small amount of data. But when plain text becomes very large, the difference becomes clear.

Encrypting as much as 80 Mega Bytes of plain text required 12.3 seconds for a normal table lookup approach whereas the same operation required 52.1 seconds for a RAT assisted approach on the same platform. So it turns out that, the proposed approach is approximately 300% slower than a normal approach. In other words, encrypting 1 Byte requires 4 times more time than a regular approach. In real time, we hardly need to encrypt bulk volume of data. Novertheless, RAT is impractical for cases where we really need to encrypt excess amount of data.

Disable Cache Sharing. To protect against software-based attacks, it would suffice to prevent cache state effects from spanning process boundaries. But practically this is very expensive to achieve. On a single threaded processor, it would require flushing all caches during every context switch. On a multi threaded processor, it would also require the logical processors to use separate logical caches, statically allocated within the physical cache; some modern processors do not support such a mode [16]. Using RAT will not impose any restriction about cache sharing which is perfectly suitable for cloud environment and virtual machines (without causing evictions and filling).

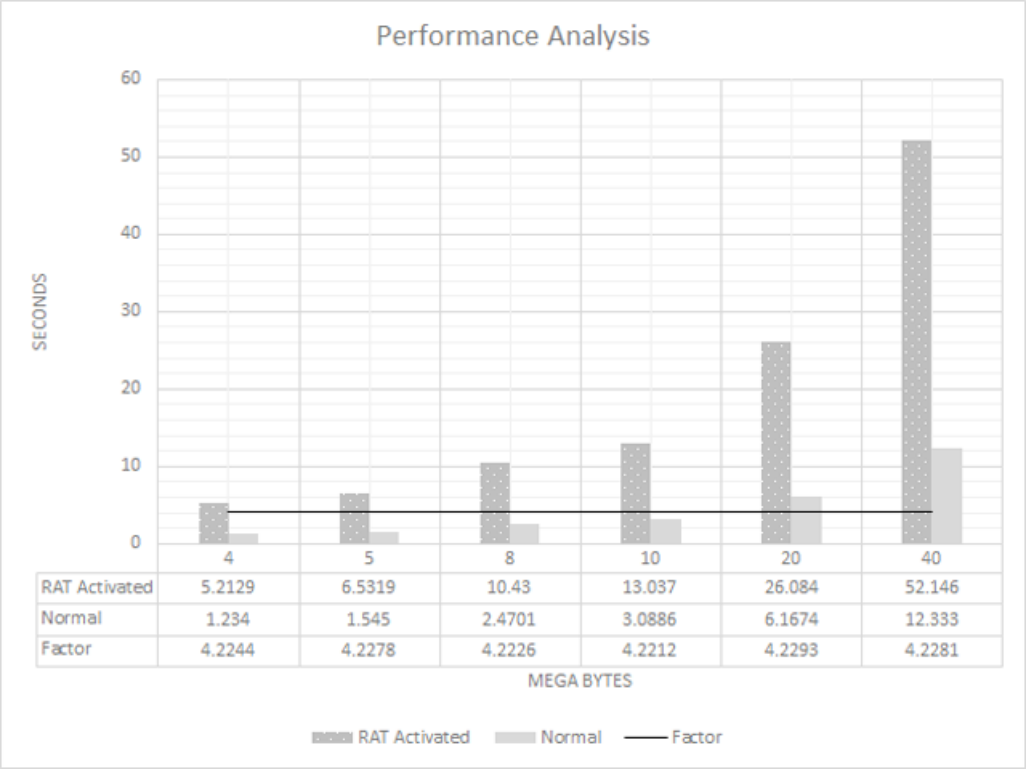

Figure 4: Performance analysis of AES implementations with (dotted column) and without (solid column) assistance of Random Address Translator in a same platform. Due to mapping overhead, a RAT assisted implementation is 4.2 times slower than a regular table lookup approach. The straight line indicates that this slowing factor doesn't depend on volume of data to be encrypted.

Static or Disabled Cache. One brutal countermeasure against the cache-based attacks is to completely disable the CPU's caching mechanism. Of course, the effect on performance would be devastating. A more attractive alternative is to activate a "no-fill" mode where the memory accesses are serviced from the cache when they hit it, but accesses that miss the cache are serviced directly from the memory. The steps are:

1. Preload the AES tables into the cache
2. Activate the "no-fill" mode
3. Perform encryption
4. Deactivate the "no-fill" mode

The section spanning (1) and (2) is critical, and attacking process must not be allowed to run during this time [16]. However, the major drawback of this approach is that, during the encryption process the overall system will be slowed down by a factor since cache access is limited during the encryption process. But using RAT, the "no-fill" mode is not required. All the other processes can be allowed to access the cache even during the encryption proceeds. To get the best result, the mapping integers should be registered.

Dynamic Table Storage. The cache-based attacks observe memory access patterns to learn about the table lookups. Instead of eliminating these, we may try to decorrelate them. For example, one can use many copies of each table, placed at various offsets in memory, and have each table lookup use a pseudorandomly chosen table. Somewhat more compactly, one can use a single table, but pseudorandomly move it around memory several times during each encryption [16]. But the major drawback of this approach is that, it will incur cache misses more than ever. This implies that the encryption process will be slowed down by a large factor. More importantly the performance and security of this approach would become architecture

dependent. Using RAT helps us alleviate this situation by obviating the need of moving around the Lookup table in the memory.

Some important benefits of using RAT are

- The Cipher Text remains the same and the AES implementation becomes a client dependent process. We don't need to worry about how other ends of the terminals are encrypting their data. As long as the key doesn't change, Cipher Text remains the same.
- No need of NOP operation. NOP operations are introduced to scramble uniform patterns in various fields. In case of cache-timing analysis, NOP operation would mean accessing Lookup table entry, but doing no operation with it.
- No need to introduce disturbance in plaintext. Sometimes plaintext is modified before it is passed to the encrypting process. This sort of approach is not suitable for large scale application where the interface must be kept as simple as possible.

7. Future Works

Intel and AMD proposed an extension to x86 instruction set named AES-NI in 2008 [24]. This extension added 7 new instructions to the instruction set with a view to abstracting the encryption process. For example, AESENC instruction would perform one round of an AES encryption flow. A performance analysis showed that AES-NI has an increase in throughput from approximately 28.0 cycles per Byte to 3.5 cycles per Byte [24]. Such initiatives are really appreciable, but these sort of instructions are not suitable for RISC processors.

Proposing a 4 times slower solution to AES implementation in pursuit of eliminating cache attack threats might not be the best way to come around. There is no point of concerning about size overhead because two 32 bit integers can even be stored in registers these days. The real bottleneck is the inverse mapping function that processes on these integers to get the correct offset to read from a perplexed lookup table. This function is called as many as 16 times during each round. Reduction of even one operation in this function can bring about a visible performance improvement. Creating a new instruction for this function can maximize the improvemnt.

References

References

[1] C. Burwick, D. Coppersmith, E. DAvignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr, L. OConnor, M. Peyravian, D. Safford, et al., Mars-a candidate cipher for aes, NIST AES Proposal 268.

[2] J. Daemen, V. Rijmen, Aes proposal: Rijndael, NIST AES Proposal.

[3] R. Anderson, E. Biham, L. Knudsen, Serpent: A proposal for the advanced encryption standard, NIST AES Proposal 174.

[4] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, Twofish: A 128-bit block cipher, NIST AES Proposal 15.

[5] N. F. Pub, 197: Advanced encryption standard (aes), Federal Information Processing Standards Publication 197 (2001) 441–0311.

[6] B. Schneier, Schneier on Security: Crypto-Gram, `https://www.schneier.com/crypto-gram-0010.html`, [Online; accessed 19-June-2014] (2000).

[7] L. Spadavecchia, A network-based asynchronous architecture for cryptographic devices (2006).

[8] R. D. D. Boneh, R. Lipton, On the Importance of Checking Cryptographic Protocols for Faults, The Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'97) (1997) 37–51.

[9] P. C. Kocher, Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems, in: Advances in Cryptology-CRYPTO'96, Springer, 1996, pp. 104–113.

[10] J. R. Rao, P. Rohatgi, Empowering side-channel attacks, IACR Cryptology ePrint Archive 2001 (2001) 37.

[11] K. Gandolfi, C. Mourtel, F. Olivier, Electromagnetic analysis: Concrete results, in: Cryptographic Hardware and Embedded Systems-CHES 2001, Springer, 2001, pp. 251–261.

[12] J.-J. Quisquater, D. Samyde, Electromagnetic analysis (ema): Measures and counter-measures for smart cards, in: Smart Card Programming and Security, Springer, 2001, pp. 200–210.

[13] J. Daemen, V. Rijmen, The design of Rijndael: AES-the advanced encryption standard, Springer, 2002.

[14] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM 21 (2) (1978) 120–126.

[15] D. A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: the case of aes, in: Topics in Cryptology–CT-RSA 2006, Springer, 2006, pp. 1–20.

[16] E. Tromer, D. A. Osvik, A. Shamir, Efficient cache attacks on aes, and countermeasures, Journal of Cryptology 23 (1) (2010) 37–71.

[17] N. Paladi, Trusted computing and secure virtualization in cloud computing, Ph.D. thesis, Luleå University of Technology (2012).

[18] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, P. Rohatgi, Efficient rijndael encryption implementation with composite field arithmetic, in: Cryptographic Hardware and Embedded SystemsCHES 2001, Springer, 2001, pp. 171–184.

[19] M. Matsui, How far can we go on the x64 processors?, in: Fast Software Encryption, Springer, 2006, pp. 341–358.

[20] M. Matsui, J. Nakajima, On the power of bitslice implementation on intel core2 processor, in: Cryptographic Hardware and Embedded Systems-CHES 2007, Springer, 2007, pp. 121–134.

[21] R. Könighofer, A fast and cache-timing resistant implementation of the aes, in: Topics in Cryptology–CT-RSA 2008, Springer, 2008, pp. 187–202.

[22] R. V. Meushaw, M. S. Schneider, D. N. Simard, G. M. Wagner, Device for and method of secure computing using virtual machines, uS Patent 6,922,774 (Jul. 26 2005).

[23] W. Stallings, Cryptography and Network Security: Principles and Practice, 5th Edition, Prentice Hall, 2011.
URL `http://books.google.com.bd/books?id=wwfTvrWEKVwC`

[24] AES instruction set, `http://en.wikipedia.org/wiki/AES_instruction_set`, accessed: 2014-12-19.