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

Refazul Islam Refat[a], Euna Islam[a], Md. Mosaddek Khan[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) that is 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 index 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 in a multiprogramming environment can lead to disastrous consequences. If the plaintext is known, recovery of half of the key by any privileged process is a matter of seconds. The

*Corresponding Author. Mob.: 011900

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

prevention is not trivial; there exists 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 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 operations triggering encryptions, in a total of 65 milliseconds. The attack required the attacker to be able to run programs on the same system or platform that is 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. Some of them even requires alteration of cipher text that might cause significant overhead in distributed systems to maintain synchronization. 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.

3.1. Memory Access in AES Implementations

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 $x^{(r)} = (x_0^{(r)}, x_1^{(r)}, ..., x_{15}^{(r)})$ at each round r. The inital state $x^{(0)}$ is computed by $x_i^{(0)} = p_i \oplus k_i (i = 0, 1, ..., 15)$.

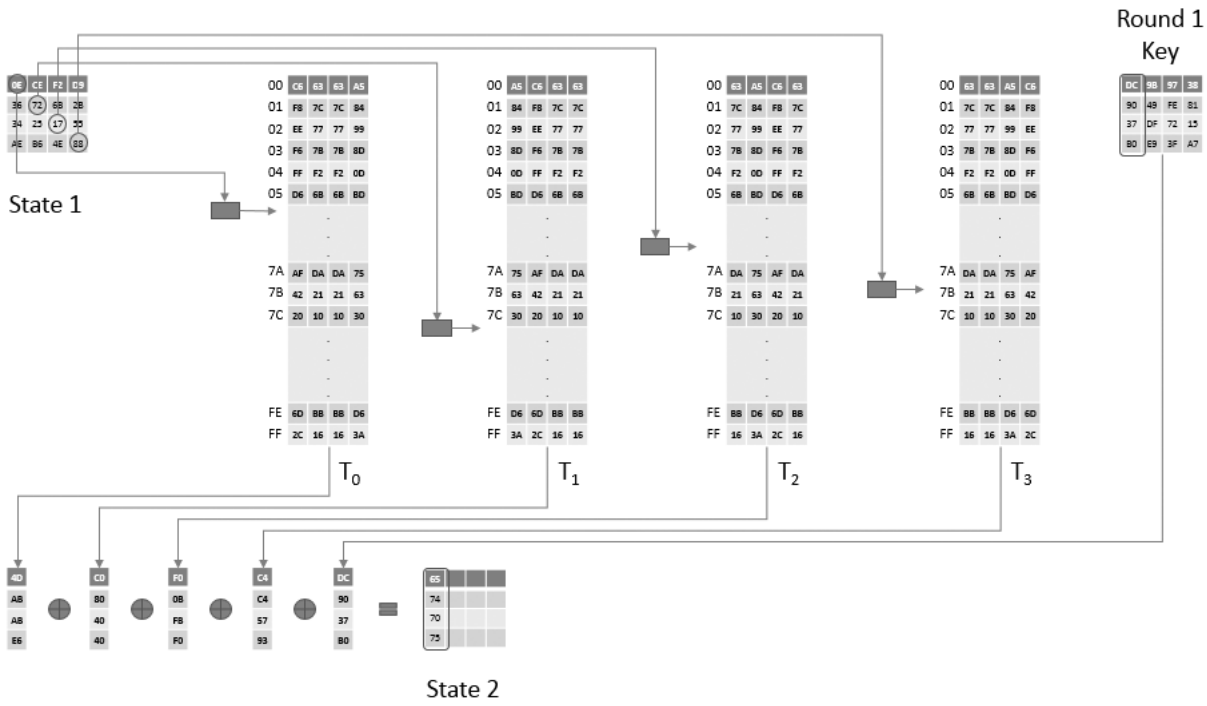

Figure 1: The figure demonstrates how $x_0^1, x_1^1, x_2^1, x_3^1$ are computed. Note that the bytes of the state matrix are being used as index of the lookup tables. Since the lookup table remains the same for a particular S-Box, an index byte will always cause lookup from the same block of cache memory

Thus, 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 formulation of AES, here the lookup tables 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* [16]. It is clear from the above discussion that computation of a state matrix requires 16 Table lookups and 16 XOR operations.

We know, the first round state matrix is computed by XOR operation between the 16-byte plain text and the 16 byte plain key. One important point to notice from the previous discussion is that the state bytes are being used as index to the Lookup tables. Thus any Information of the accessed index will directly translate to Encryption Key Byte. For example, lets say we know $x_i^{(0)}$. Since plaintext is known for triggered encryptions, we know $x_i^{(0)}$ too.

$$x_i^{(0)} \leftarrow x_i^{(0)} \oplus k_i^{(0)}$$

This implies,

$$k_i^{(0)} \leftarrow x_i^{(0)} \oplus x_i^{(1)}$$

Thus knowing $x_i^1(1)$ for $i = 0, 1, 2, ..., 15$ is sufficient to recover the full encryption key under the assumption that the plaintext is known. This is the weakest part of Round 1; in fact the weakest part of the Cipher.

3.2. How Lookup Tables fit in Memory and Cache

Lets turn our attention to how AES Lookup tables fit in Memory and in Cache. We know, each Lookup table has 256 4-Byte entry. That gives it a total size of $256 \cdot 4 = 1024$ Bytes. Now, typical size of *memory block* is 64 Bytes even on modern microprocessors. So for simplicity, we will assume the *memory block* size B to be 64 Bytes throughout this paper. Therefore, each table T_i will occupy 16 memory blocks.

Also assume that, the initial address of T_i congruent to the cache. That is, 1st block of the table will be cached in one of the W *cache lines* in the 1st *cache set*.

4. The Idealized Prime+Probe Technique

The 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.

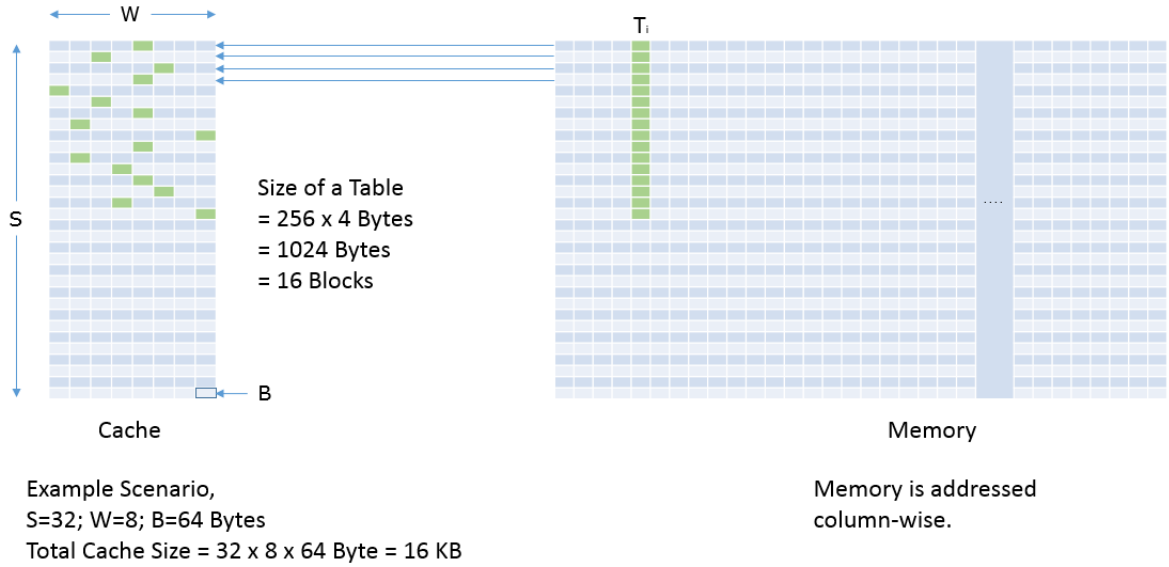

Figure 2: Demonstration of how AES Lookup Table fits in memory and cache. On the left side, number of rows implies total number of cache sets, whereas, number of columns implies associativity of the cache. On the right side, instead of depicting memory as a continuous array of blocks, it is portrayed in congruence with cache memory. Note that all blocks in a row of memory competes for the blocks of corresponding row in cache memory.

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. If $S = 32$, $W = 8$ and $B = 64$ Bytes, then

$$\text{Total size of the cache} = S \cdot W \cdot B = 16 \text{ KB}$$

As implied by the figure above, performing a serial read on a contiguous memory as big as the cache results in inception of filling the cache. Continuing this way, we'll end up filling the cache completely after the reading is finished.

Trigger. With the Cache filled with our data, we're in a position to invoke the encrypting process and trigger an encryption of some known plain text. As the encrypting process reaches the First Round and performs the first lookup, it is interrupted. An important point to note here is, we're idealizing the model to illustrate the susceptibility of the First Round. Recovering the AES Encryption Key is not the actual intent of this paper.

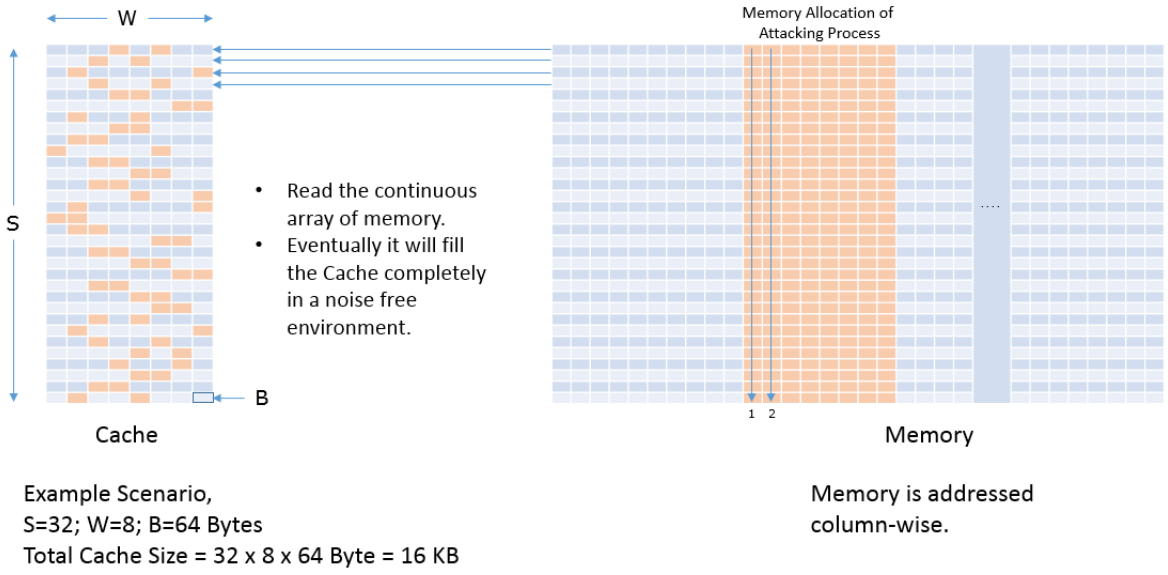

Figure 3: Demonstration of the Cache filling process. Note that reading continuous array of memory will cause allocation of one block from each cache set and due to Principle of Locality, immediate next read cycle will cause allocation of another block (different from the previous) from each cache set.

Probe. After we interrupt the encrypting process just after it performed its first lookup during the First Round of AES, we perform the Prime again, i.e. we fill the cache again. But this time, we maintain a clock measuring of how long it takes to read a *memory block*. Before the Trigger phase, the cache was completely filled by our data. But after that Trigger phase, one block of the Lookup table T_0 is now in the cache due to the lookup operation. So all but that block will suffer from a cache miss.

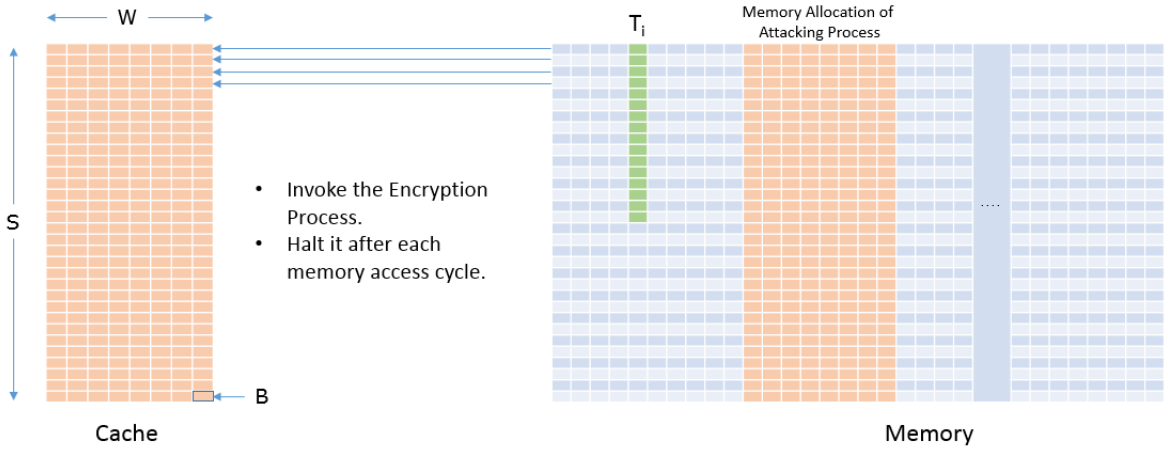

Figure 4: Demonstration of the Prime phase where reading a continuous array of memory equal to the size of the cache will eventually fill the cache completely with attacker's data in a noiseless environment.

Observing the position of the cache miss, we can easily identify which block of T_0 was accessed since we assumed Lookup Tables T_i to be congruent with the cache. Now if we can identify which block of T_0 was accessed during the First Round, we can extract some valuable piece of information. Since size of *memory block* is 64 Bytes, each *memory block* holds 16 entry of the Lookup Table. So for example, if 7th block of T_0 was accessed, we can conclude the index $x_0^{(1)}$ was somewhat between 0x60 and 0x6F, i.e. high nibble is 0x6. We know

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

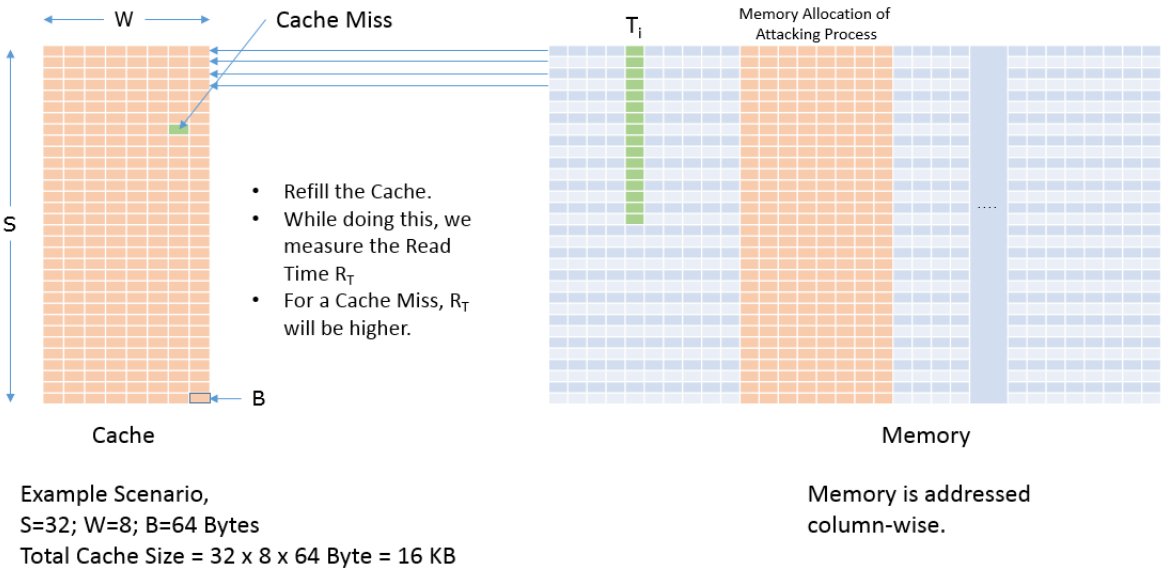

Figure 5: Demonstration of how information about the location of the cache miss during first round reveals critical information about the encryption key. In the example scenario, the high nibble of a Byte (0x6) is easily gleaned.

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:

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

Because we know $x_0^{(0)}$ from the plain text and the high nibble of $x_0^{(1)}$ from the Prime+Probe technique, we 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 $i = 0, 1, 2, ..., 15$, we can calculate $k_i^{(0)}$ for $i = 0, 1, 2, ..., 15$ and dig up half of the Encryption Key. The idealized procedure is summarized in algorithm 1.

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
```

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 will 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.

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.

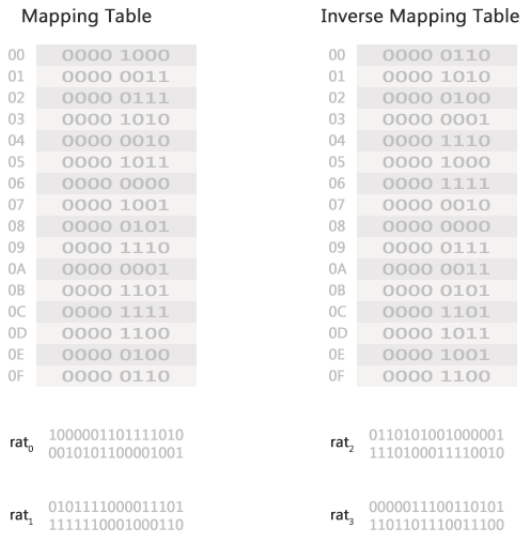

Figure 6: The figure illustrates the state of the mapping and inverse mapping tables after randomization. Since numbers from 0 to 15 requires only 4 bits and a Byte is capable of storing 8 bits, the tables can be condensed by a factor of two and can be conveniently stored as a set of four 32 bit integers rat_i for $0 <= i <= 3$

The following algorithm outlines the process. It takes no parameters and returns a set of four integers. The first two integers rat_0 and rat_1 will be used randomize the Lookup table and the last two integers rat_2 and rat_3 will be used to retrieve correct piece of information from the Lookup table.

Now that we have the mapping integers ready, we can proceed to randomize the Lookup table. But for that we need to operate on the integers (rat_1 and rat_2). The following algorithm does that operation. It takes an offset Byte as argument and returns another offset Byte.

To further 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 make the process really fast. After the randomization is done, that temporary table is completely removed from memory.

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 comes 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.

6. Performance Evaluation

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.

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 3 Mapping Fuction

function MAP(i)
  if $((i \ \& \ 128) = 0)$ **then** ▷ & resembles bitwise and operation
    $j \leftarrow 0$
  end if
  $k \leftarrow ((255 - i) \gg 4) \ll 2$ ▷ $\ll$ and $\gg$ resemble bitwise left and right shift operations respectively
  $x \leftarrow (rat_j \gg k) \ \& \ 15$
  $y \leftarrow i \ \& \ 15$
  return $(x \ll 4) + y$
end function

Algorithm 4 Mingling Function

function MINGLE(table) ▷ Takes unmodified lookup table as parameter
  for $i \leftarrow 0$ to 255 **do**
    for $j \leftarrow 0$ to 3 **do**
      $temp_{i,j} \leftarrow table_{map(i),j}$
    end for
  end for
  for $i \leftarrow 0$ to 255 **do**
    for $j \leftarrow 0$ to 3 **do**
      $table_{i,j} \leftarrow temp_{i,j}$
    end for
  end for
  delete *temp*
  return *table*
end function

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
```

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. The SubBytes Transformation phase requires 16 table lookups for the 16 Bytes of a State matrix. The ShiftRows Transformation phase requires 16 shift operations. The MixColumns Transformation phase requires 8 shift operations, 8 conditional XOR operations and 12 mandatory XOR operations. Finally the AddRoundKey Transformation phase requires 16 XOR operations.

To summarize the fact, we need a total of 24 shift operations, 16 table lookups, 8 conditional XOR operations, 28 mandatory XOR operations for a single round. This is demonstrated in figure 10.

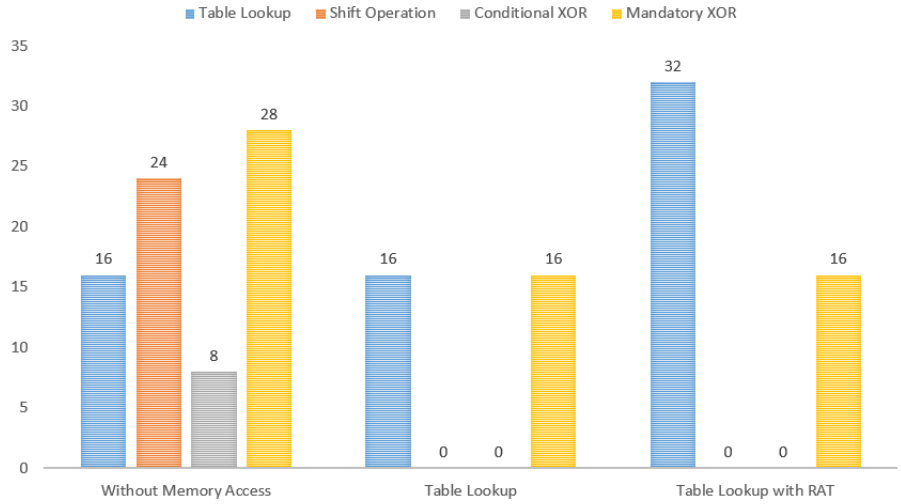

Figure 7: Comparison of RAT with straightforward AES implementations in a single round. An implementation that completely avoids memory access requires 16 table lookups, 24 shift operations and 36 XOR operations (8 of which them are conditional), whereas introduction of RAT incurs 2 times more table lookups and 16 XOR operations in a single round.

In fact, the conditional XOR operation is vulnerable to timing attack too [23]. Other than lagging behind in terms of security, straightforward implementation required more computation time. So avoiding memory access doesn't save the day. 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 maintaining two table. The Lookup table requires 4096 Bytes of memory space and RAT requires another 1024 Bytes of space.

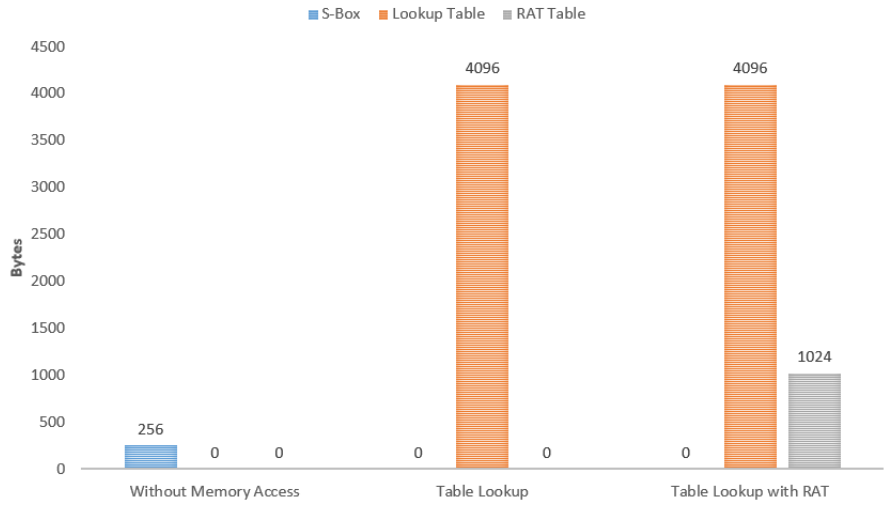

Figure 8: Performance Evaluation of RAT in terms of Memory Usage.

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 processor with simultaneous multi threading, 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).

Static or Disabled Cache. One brutal countermeasure against the cache-base 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

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

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 factor. More importantly the performance and security of this approach are very architecture dependent. Using RAT helps us alleviate this situation by obviating the need of moving around the Lookup table in the memory.

6.1. Other Major Advantages

Some important benefits of using the 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

- There are some limitations in the process of constructing the RAT table. One major limitation is that swapping is not allowed more than once for an entry. Thus there can be a maximum of 128 swaps for a single Lookup table. To allow more number of swaps per entry, more sophisticated data structure is required.
- The big picture that summarises the performance difference with the straightforward Table lookup approach is given below

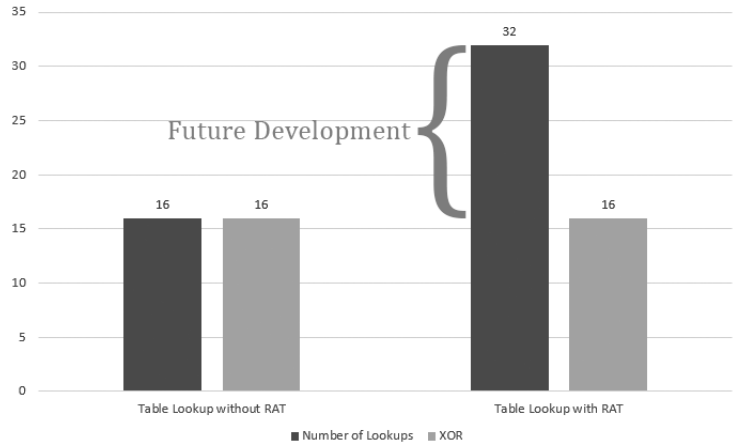

Figure 9: Comparison with straightforward Table Lookup approach.

As we can see, we need 2 times more table lookups than the ordinary approach. The area of future development is to reduce this factor.

- Another big concern is that, RAT tables add a fair amount of memory overhead to the encrypting process. Whereas the Lookup tables altogether require 4KB of space, RAT tables requires 1KB of space. Although not a big figure to worry about, space overhead can be reduced with the assistance of some more tables of smaller sizes.

8. Conclusion

Side channel attacks don't divulge everything overnight; instead they ease the calculations and assumptions that helps rest of the operations accomplished in bounded time. Although the existing AES encryption model has sufficient complexity to blow away any straightforward attempt to break the security, side-channels still poses a real threat and a bad implementation of the cipher might lead to dire consequences.

The most challenging part of security is now making the implementation of the existing ciphers resistant to side-channel attacks. Preventing leakage of information through side-channel is not a trivial task at all. The experiment has shown that even the straightforward Table lookup approach can be made secure without the assistance of any piece of special software or hardware. And that might be the most important requirement in the industry.

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`