Design, Implementation and Performance Analysis of Highly Efficient Algorithms for AES Key Retrieval in Access-driven Cache-based Side Channel Attacks[*]

Ashokkumar C, M. Bhargav Sri Venkatesh[†], Ravi Prakash Giri , Bernard Menezes
Department of Computer Science & Engineering
Indian Institute of Technology, Bombay, INDIA
{*ashokkumar, raviprakash, bernard*}*@cse.iitb.ac.in, sri@ee.iitb.ac.in*

***Abstract*—Leakage of information between two processes sharing the same processor cache has been exploited in many novel approaches targeting various cryptographic algorithms. The software implementation of AES is an especially attractive target since it makes extensive use of cache-resident table lookups. We consider two attack scenarios where either the plaintext or ciphertext is known. We employ a multi-threaded spy process and ensure that each time slice provided to the victim (running AES) is small enough so that it makes a very limited number of table accesses. We design and implement a suite of algorithms to deduce the 128-bit AES key using as input the set of (unordered) cache line numbers captured by the spy threads in an access-driven cache-based side channel attack. Our algorithms are expressed using simple relational algebraic operations and run in under a minute. Above all, our attack is highly efficient – we demonstrate recovery of the full AES key given only about 6–7 blocks of plaintext or ciphertext (theoretically even a single block would suffice). This is a substantial improvement over previous cache-based side channel attacks that require between 100 and a million encryptions. Moreover, our attack supports varying cache hit/miss observation granularities, does not need frequent interruptions of the victim and will work even if the victim makes up to 60 cache accesses before being interrupted. Finally, we develop analytic models to estimate the number of encryptions/decryptions required as a function of access granularity and compare model results with those obtained from our experiments.**

***Index Terms*—AES, access-driven, cache attacks, side channel, table lookup**

1. Introduction

Through much of the history of cryptography, attacks on cryptographic algorithms have focused on cracking hard mathematical problems such as the factorization of very large integers (which are the product of two very large primes) and the discrete logarithm problem [1]. More recently, however, side channel attacks have gained prominence. These attacks leak sensitive information through physical channels such as power, timing, etc. and typically are specific to the actual implementation of the algorithm [2]. An important class of timing attacks is that based on obtaining measurements from cache memory systems.

The Advanced Encryption Standard (AES) [3], a relatively new algorithm for secret key cryptography, is now ubiquitously supported on servers, browsers, etc. Almost all software implementations of AES including the widely used cryptographic library, OpenSSL, make extensive use of table lookups in lieu of time-consuming mathematical field operations. Cache-based side channel attacks aim to retrieve the key of a victim performing AES by exploiting the fact that access times to different levels of the cache-main memory hierarchy vary by 1–2 orders of magnitude.

Cache-based side channel attacks belong to one of three categories. Timing-driven attacks measure the time to complete an encryption [4]. Trace-driven attacks create profiles of a cache hit or miss for every access to memory during an encryption [5]. Finally, access-driven attacks need information only about which lines of cache have been accessed, not their precise order. Two of the most successful access-driven attacks [6], [7] belong to the last category.

We consider two possible scenarios. In the first (Scenario I), a victim process runs on behalf of a data storage service provider who securely stores documents from multiple clients and furnishes them on request after due authentication. The same key or set of keys is used to encrypt documents from different clients prior to storage. In Scenario II, two entities, A and B, exchange encrypted messages. The victim, on B's machine, decrypts blocks of ciphertext received from A.

In both scenarios, we assume that the attacker or spy is hosted on the same processor core as the victim and that their executions are interleaved as in [7]. Moreover, both attacker and victim use the OpenSSL library. So only a single copy of OpenSSL is resident in main memory and is mapped to the virtual spaces of both, attacker and victim.

The spy process flushes out all the cache lines containing the AES tables. When the victim is scheduled, it brings in some of the evicted line(s). When control returns back to the spy, it determines which of the evicted lines were fetched

[*]*A version of this paper is to be published in the Proceedings of the IEEE European Symposium on Security and Privacy, March 2016, Saarbrücken, Germany.*
[†]3rd *year B.Tech student in Department of Electrical Engineering, Indian Institute of Technology, Bombay.*

by the victim by measuring the time to access them. It then flushes out the AES tables from cache before relinquishing control of the CPU.

Our algorithms to deduce the AES key use two crucial inputs. The first of these is either the plaintext or the ciphertext. In Scenario I, the attacker could pose as a customer to the data storage service provider and request that his documents (plaintext) be securely stored. The attack in Scenario II makes the reasonable assumption that the ciphertext to B can be eavesdropped upon.

The second crucial piece of information is the set of line (or block) numbers of AES table entries accessed by the victim. Several entries in the AES table are placed on a single cache line and the espionage network we employ provides a set (not list) of lines accessed. The absence of spatial information (the specific table entry on a line) and temporal information (the order of accesses) makes it challenging to deduce the key especially for sets with larger cardinalities. Our espionage software was ported on Intel® Core 2 Duo, Core i3 and Core i5 with hardware support for AES turned off in the latter two cases.

Given the above two inputs, our main contribution is the design and implementation of a suite of algorithms to deduce the AES key. Our algorithms are simple and are elegantly expressed using relational algebraic operations. Even unoptimized versions of our algorithms in Python run in under a minute. Above all, our attack is highly efficient – we demonstrate recovery of the full 128-bit AES key given only about 6–7 blocks of plaintext or ciphertext (theoretically even a single block would suffice). This is a substantial improvement over previous cache-based side channel attacks that require between 100 and a million encryptions. Moreover, our attack requires preemption of the victim only 5-7 times per encryption/decryption and will work even if the victim makes up to 60 accesses before being interrupted. Finally, we develop analytic models to estimate the number of encryptions/decryptions required and compare model results with those obtained from our experiments.

This paper is organized as follows. Section 2 contains a brief introduction to AES, its implementation using lookup tables and processor cache. Section 3 describes our espionage setup including the spy controller and spy threads. Sections 4 and 5 present the algorithms, model and results related to the attacks in Scenarios I and II respectively. Section 6 discusses other issues of relevance, limitations and countermeasures. Section 7 summarizes work related to the theme of this paper and Section 8 contains the conclusions.

2. Preliminaries

We first summarize the software implementation of AES and then introduce the basics of cache memories.

2.1. AES Summary

AES is a symmetric key algorithm standardized by the U.S. National Institute of Standards and Technology (NIST) in 2001. Its popularity is due to its simplicity yet it is resistant to various attacks including linear and differential cryptanalysis. The full description of the AES cipher is provided in [3]. Here, we mainly focus on its software implementation.

AES is a substitution-permutation network. It supports a key size of 128, 192 or 256 bits and block size = 128 bits. A round function is repeated a fixed number of times (10 for key size of 128 bits) to convert 128 bits of plaintext to 128 bits of ciphertext. The 16-byte input or plaintext $P = (p_0, p_1, ..., p_{15})$ may be arranged column wise in a 4×4 array of bytes. This "state array" gets transformed after each step in a round. At the end of the last round, the state array contains the ciphertext.

All rounds except the last involve four steps – Byte Substitution, Row Shift, Column Mixing and a round key operation (the last round skips the Column Mixing step). The round operations are defined using algebraic operations over the field $GF\left(2^8\right)$. For example, in the Column Mixing step, the state array is pre-multiplied by the matrix B given below.

$$B = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

Each element in B is an element of $GF\left(2^8\right)$ represented as 2 hexadecimal characters. The original 16-byte secret key $K = (k_0,\ k_1,\ ...,\ k_{15})$ (arranged column wise in a 4×4 array of bytes) is used to derive 10 different round keys to be used in the round key operation of each round. The round keys are denoted $K^{(r)}$, $r = 0, 1...9$.

In a software implementation, field operations are replaced by relatively inexpensive table lookups thereby speeding encryption and decryption. In the versions of OpenSSL targeted in this paper, five tables are employed (each of size 1KB). A table T_i, $0 \leq i \leq 4$ is accessed using an 8 bit index resulting in a 32-bit output.

Let $x^{(r)} = \left(x_0^{(r)}, \ldots, x_{15}^{(r)}\right)$ denote the input to round r (i.e. the state array at the start of round r). The initial state $x^{(0)} = \left(x_0^{(0)}, \ldots, x_{15}^{(0)}\right)$ is computed by $x_i^{(0)} = p_i \oplus k_i$ for $i = 0, \ldots, 15$. The output of round r, $r = 0, 1, ..., 8$ is obtained from the input using 16 table lookups and 16 XOR operations as shown below (1) – (4).

$$\left(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}\right) \leftarrow T_0\left[x_0^{(r)}\right] \oplus T_1\left[x_5^{(r)}\right] \oplus T_2\left[x_{10}^{(r)}\right] \oplus T_3\left[x_{15}^{(r)}\right] \oplus K_0^{(r)} \tag{1}$$

$$\left(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}\right) \leftarrow T_0\left[x_4^{(r)}\right] \oplus T_1\left[x_9^{(r)}\right] \oplus T_2\left[x_{14}^{(r)}\right] \oplus T_3\left[x_3^{(r)}\right] \oplus K_1^{(r)} \tag{2}$$

$$\left(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}\right) \leftarrow T_0\left[x_8^{(r)}\right] \oplus T_1\left[x_{13}^{(r)}\right] \oplus T_2\left[x_2^{(r)}\right] \oplus T_3\left[x_7^{(r)}\right] \oplus K_2^{(r)} \tag{3}$$

$$\left(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}\right) \leftarrow T_0\left[x_{12}^{(r)}\right] \oplus T_1\left[x_1^{(r)}\right] \oplus T_2\left[x_6^{(r)}\right] \oplus T_3\left[x_{11}^{(r)}\right] \oplus K_3^{(r)} \tag{4}$$

Here $K_i^{(r+1)}$ refers to the i^{th} column of $K^{(r+1)}$.

To compute the last round, Table T_4 is used instead of $T_0, ..., T_3$. Due to the absence of the Column Mixing step, the value returned by the table lookup is XORed with the corresponding byte of the round key. Since each round involves 16 table accesses, a complete encryption involves a total of 160 table accesses. Table 1 summarizes the notations that appear in this paper.

2.2. Cache Basics

All modern processors have multiple levels of cache intended to bridge the latency gap between main memory and the CPU. Of the machines targeted in this paper, Core 2 Duo has two levels of cache (L1 comprises 32KB D-cache and 32 KB I-cache and L2 is 2MB) while Core i3 has three levels (L1 comprises 32KB I-cache and 32KB D-cache, each core has a private 256 KB L2 cache and a shared 3MB L3 cache).

The granularity of data transfer between different levels of cache is a block or line. On all the machines we used, the line size $= 64$ bytes. The lines of a cache are grouped into sets – a line from main memory is mapped to exactly one set though it may occupy any position in that set. The number of lines in a set is the associativity of the cache. In the machines we worked with, L1 and L2 caches are 8-way set associative while L3 is 12-way set associative in Core i3 and i5.

During the encryption/decryption of a block of plaintext/ciphertext, parts of the AES tables are brought into cache as needed. Each entry in a table is 4 bytes, so 16 entries can be accommodated in a single line. Each table contains 256 entries, so a table fits into 16 blocks. The first four bits of an 8-bit table index identify a line within the table while the last four bits specify the position of the entry within the line. Thus, the first four bits of a table index are leaked if the attacker can determine which line of the cache was accessed.

3. Espionage Setup

Our espionage network borrows from the work in [8]. It comprises a Spy Controller (SC) and a Spy Ring. The SC runs on one CPU core while the ring of spy threads runs on another core together with the victim. The executions of spy threads and the victim (V) are interleaved. We refer to an execution instance of V as a run. During each run, V accesses the AES tables and the next spy thread that is scheduled attempts to determine which lines of the table were accessed in the preceding run of V. The default time slice (or quantum) assigned by the OS to a process is large enough to accommodate thousands of cache accesses. But if V is given this full time slice, it would perform tens of encryptions (each encryption involves 160 table accesses) thus making it impossible to obtain any meaningful information about the encryption key.

TABLE 1: Notations

Notation	Explanation
K, $K^{(i)}$	AES Key or i^{th} round key represented as 4x4 byte array (column wise)
k_i	i^{th} byte of AES key
P	128-bit plaintext represented as 4x4 byte array
p_i, $p_{i,e}$	i^{th} byte of plaintext (in encryption e)
C	128-bit ciphertext represented as 4x4 byte array
c_i, $c_{i,e}$	i^{th} byte of ciphertext (in encryption e)
T_i	AES Table, $0 \leq i \leq 4$
B	Column Mixing operation matrix
ρ	Set of line numbers of AES tables accessed in a single execution quantum or run of the victim. The cardinality of this set is referred to as the run size.
$\rho_{x-y,t,e}$	Set of line numbers of AES table T_t accessed in run x to y of encryption e
x'	High-order nibble of byte x
x''	Low-order nibble of byte x
r_i	Relation containing various AES subkey attributes
$r_i \bowtie r_j$	Join of r_i and r_j
$\sigma_{i \cdot j}(r)$	All tuples in relation r satisfying Equation-i with plaintext used in the j^{th} block encryption
$r_i \times r_j$	Cartesian product of relation r_i and r_j
ε	Number of encryptions required to retrieve AES key
δ	Number of decryptions required to retrieve AES key
c	Compression ratio (ratio of cardinalities of output to input relations of a select operation)

We exploit the fact that the Completely Fair Scheduler (CFS) employed in many Linux versions uses a calculation based on virtual runtimes to ensure that the aggregate CPU times allocated to all processes and threads are nearly equal. If the number of threads is n and they execute in round-robin fashion, then each run of V is roughly of duration x/n where x is the uninterrupted time allocated to any running thread.

The task of a spy thread is to measure the access time of each cache line containing an AES table and then flush the tables from all levels of cache. It then signals the SC through a shared boolean variable, *finished*, that its task is complete and blocks on the condition myID = nextThreadID. At this point, all spy threads are in the blocked state and the OS resumes execution of V.

The SC continuously polls the *finished* flag. When it finds that *finished* has been set to true, it signals the spy thread that has waited the longest and sets *finished* to false.

Our attacks were implemented on Intel® Core 2 Duo E4500, 2.20GHz processor running Debian 8.0, 32-bit, kernel version 3.16. We also experimented with Intel® Core i3 2100, 3.10GHz processor running Debian 8.0, 32-bit, kernel version 3.16 and Core i5 2540M, 2.60GHz processor running Debian Kali Linux 1.1.0, 64-bit, kernel versions

Spy Thread

```
while true do
    conditionWait until myID = nextThreadID
    for each cacheLine containing AES tables do
        if accessTime $<$ THRESHOLD then
            isAccessed[cacheLine] $\leftarrow$ true
        end if
        clflush(cacheLine)
    end for
    lock(mutex)
        finished $\leftarrow$ true
    unlock(mutex)
end while
```

Spy Controller

```
while true do
    while finished $\neq$ true do
    end while
    signal(nextThreadID)
    lock(mutex)
        finished $\leftarrow$ false
    unlock(mutex)
end while
```

3.14 and 3.18. The C implementation of OpenSSL versions 0.9.8a and 1.0.2a were employed.

The third component of the espionage infrastructure is the Analytics Engine. Based on the inputs from the spy threads and the known plaintext or ciphertext, it recovers the original AES key as explained in the remainder of this paper.

4. Scenario I Attack

We first present the First Round Attack wherein the high-order nibble of each of the 16 bytes of the AES key are obtained. These serve as input to the Second Round Attack described later.

4.1. First Round Attack - Description

In Scenario I, the same key is used to encrypt several blocks of plaintext that are known to the attacker. As explained in section 2, the elements of the 4×4 input matrix to a round in AES are indices to the AES lookup tables. During the first round, this matrix is $P \oplus K$ where P is the plaintext and K is the AES key. Based on measured cache access times, a spy thread identifies which lines of the AES tables were accessed. These 4-bit "table line numbers" are the high-order nibble of each byte in the 4×4 matrix. So, the high-order nibbles of all bytes of K may be obtained.

During the first nine rounds, the victim accesses the lookup tables in round-robin fashion - T_0, T_1, T_2, T_3, T_0, $T_1, \ldots$ If the spy threads could provide the precise sequence of lines accessed, we could unambiguously deduce $k_0^{'}$, $k_5^{'}$, $k_{10}^{'}$, $k_{15}^{'}$, $k_4^{'}, \ldots$ However, a spy thread provides a set (not list) of table line numbers accessed during the preceding run.

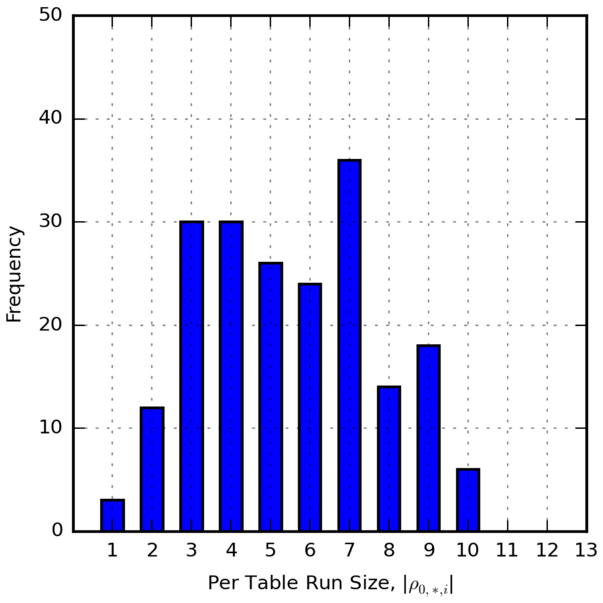

Figure 1: Histogram of per table run size of first run, $|\rho_{0,*,i}|$, in 200 encryptions

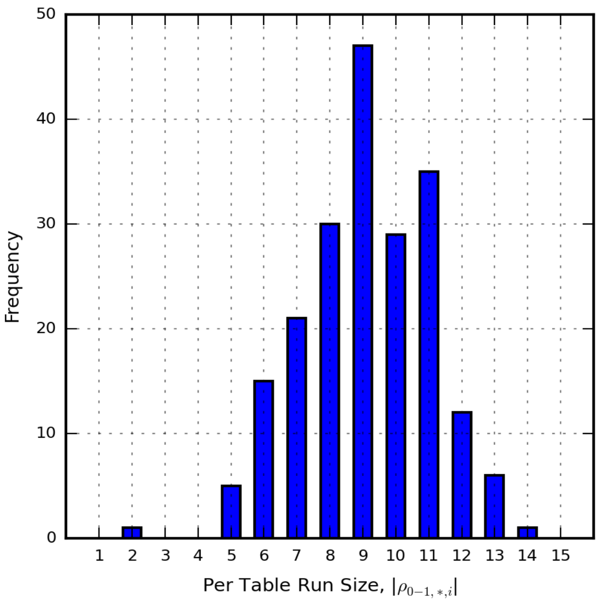

Figure 2: Histogram of per table combined run size of first two runs, $|\rho_{0-1,*,i}|$, in 200 encryptions

In addition to the set of accesses within a single run, our algorithms may require the set of all accesses made during two or more consecutive runs. Let $\rho_{x-y,t,e}$ denote the set of distinct lines accessed in runs x through y made to Table T_t in the e^{th} block encryption. The cardinality of the set of lines denoted $|\rho_{i,*,*}|$ varies across runs. As it turns out,

$|\rho_{i,*,*}|$ is an important experimentally determined parameter that impacts the performance of our algorithms. Histograms of $|\rho_{0,*,*}|$ and $|\rho_{0-1,*,*}|$ are of special relevance and are shown in Figures 1 and 2 respectively.

We assume that the victim encrypts one block after another, so it is necessary to determine when the encryption of a new block has begun. In the 5-table implementation of AES, this is straightforward since the last round accesses T_4 exclusively. Experimental results on our setup indicate that there are almost always two (or more) consecutive runs containing accesses to T_4. If the last of these runs also contains accesses to T_0, we can be sure that the encryption of the next block of plaintext has begun. We next introduce an algorithm to recover the high-order nibble of each byte of the AES key.

Algorithm 1 First Round Attack

```
Input:
    $\varepsilon$ blocks of plaintext, $\rho_{0,t,e}$ ,
    $\rho_{0-1,t,e}$ , $0 \le t \le 3$ , $1 \le e \le \varepsilon$
Output:
    High-order nibble of each byte of AES key
1:  for each table, $T_t$, $t = 0, 1, 2, 3$ do
2:      for each column index $i$,
        referencing matrix P, $0 \le i \le 3$ do
3:          // Prepare histogram for key nibble, $k'_{t+4i}$
4:          for each encryption, $e$, $e \le \varepsilon$ do
5:              if ( $|\rho_{0,t,e}| \ \ge \ 4$ ) then
6:                  $\rho = \rho_{0,t,e}$
7:              else
8:                  $\rho = \rho_{0-1,t,e}$
9:              end if
10:             for each $x \in \rho$ do
11:                 increment $histogram_{t+4i}[x \oplus p'_{t+4i,e}]$
12:             end for
13:         end for
14:     end for
15: end for
```

In total 16 histograms are created, one per high-order nibble of each byte of the AES key. $histogram_{t+4i}$ displays scores for each of the 16 possible nibble values for nibble k'_{t+4i} of the key. Since $P \oplus K$ is the input to the first round of AES, $\rho_{0,0,1}$ will contain $(p_0 \oplus k_0)'$ and possibly, $(p_4 \oplus k_4)'$, $(p_8 \oplus k_8)'$ and $(p_{12} \oplus k_{12})'$ depending on the cardinality of $\rho_{0,0,1}$. A spy thread provides these values but not necessarily in the order we would like. So starting with $\rho_{0,0,1}$, for each value x in $\rho_{0,0,1}$, Algorithm 1 increments the value at $x \oplus p_{0,0}$ in $histogram_0$. It repeats this for each encryption with $\rho_{0,0,e}$, $e \le \varepsilon$ and associated plaintext. Note that the correct key nibble in the histogram receives a boost but so do several others (which constitute false positives). As ε increases, all but the correct nibble value will fail to be incremented at least once, so the true value of the key nibble will stand out.

This procedure is repeated for each of the remaining 15 histograms. The algorithm terminates after ε is large enough that we discover the true values of all high-order key nibbles in their respective histograms.

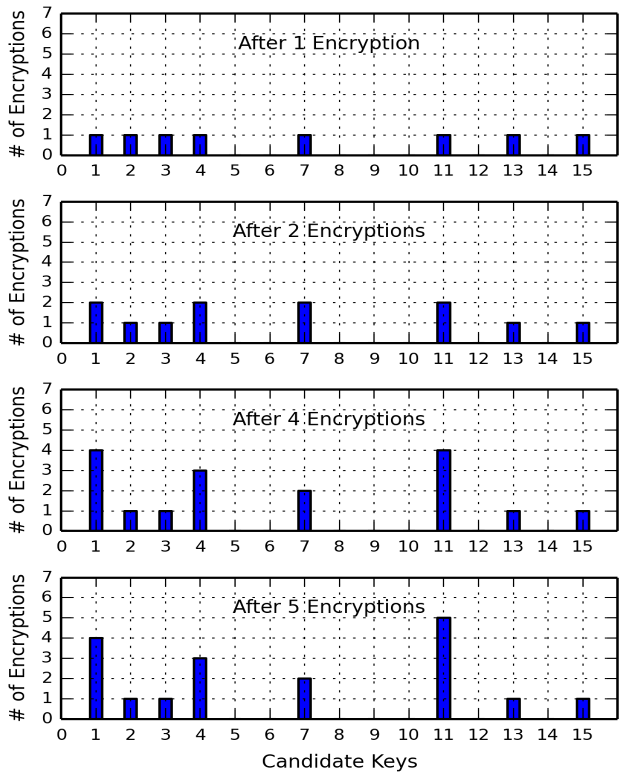

Figure 3: Evolution of histogram of score of potential high-order nibble values for key byte k_0

4.2. First Round Attack - Results

We experimentally obtained a distribution of ε values by generating 100 round keys. For each key, we generated 25 random plaintext blocks and encrypted those blocks with the key. We refer to a key together with the 25 blocks of the plaintext as a sample. We determined the number of block encryptions required to unambiguously deduce the high-order nibbles of all 16 key bytes for each sample. We found that 70 samples required between 5 and 7 encryptions, 16 samples required 8 encryptions, 13 samples required 9 encryptions and one sample required 13 encryptions.

Figure 3 shows the evolution of a specific histogram after 1, 2, 4 and 5 encryptions. It is clear that four encryptions do not suffice and the true nibble value is known only after 5 encryptions.

We next derive an expression for the average number of encryptions required to obtain the high-order nibble of each byte of the AES key. Given the plaintext for the i^{th} encryption, the probability that an incorrect value gets a boost is $\frac{|\rho|-1}{15}$. The probability that the incorrect value does not receive a boost in at least one of e encryptions is $1 - (\frac{|\rho|-1}{15})^e$. There are 16 histograms, each contains 16 possible values. Of them, a total of 240 values are incorrect. Let P_e denote the probability that each of 240 incorrect values does not receive a boost in at least one of e encryptions. So, $P_e = (1 - (\frac{|\rho|-1}{15})^e)^{240}$. The average number of encryptions required to deduce the high-order nibble of each byte of the AES key is therefore

$$\sum_{e=1}^{\infty} e \cdot (P_e - P_{e-1})$$

Figure 4 shows the average number of encryptions required to deduce the high-order nibbles of the AES key as function of the per table run size, $|\rho|$.

4.3. Second Round Attack - Description

The goal of the Second Round Attack is to obtain the low-order nibble of each byte of the AES key. For this, we use (5) – (20) which relate the second round inputs to the plaintext and to various bytes of the key. The equations are easily derived by tracking how the input to Round 1 gets transformed after Byte Substitution, Row Shift, Column Mixing and the round key operations.

We choose different subsets of the low-order nibbles of the key, say $k_0^{''}$, $k_5^{''}$, $k_{10}^{''}$, $k_{13}^{''}$, $k_{15}^{''}$ and create relations (or tables) for each subset of interest. A relation for the first subset has attributes (or columns) named $k_0^{''}$, $k_5^{''}$, $k_{10}^{''}$, $k_{13}^{''}$, $k_{15}^{''}$. During the execution of the key retrieval algorithm, each row or tuple of the relation will contain a potential subkey value. New tables may be created and are operated upon resulting in gradually reduced cardinalities so that when the algorithm terminates only the actual key or a small number of potential key values survive.

Our algorithm is best described in terms of relational algebraic operations [9] – select, Cartesian product and join. The select operator (σ) filters out tuples of a relation based on a certain predicate, i.e., only those tuples survive that satisfy the predicate. The select operation used in this work takes the form $\sigma_{i \cdot j}(r)$. The predicate $i \cdot j$ is short for "all tuples (subkeys) in relation r satisfying Equation i, $5 \leq i \leq 20$, with plaintext used in the j^{th} block encryption". Satisfiability of the predicate is limited to checking for equality of the high-order nibble values of the LHS and RHS of Equation i.

$$\begin{aligned} x_0^{(1)} &= 2 \bullet s(p_0 \oplus k_0) \oplus 3 \bullet s(p_5 \oplus k_5) \oplus s(p_{10} \oplus k_{10}) \\ &\oplus\ s(p_{15} \oplus k_{15}) \oplus s(k_{13}) \oplus k_0 \oplus 1 \end{aligned} \tag{5}$$

$$\begin{aligned} x_1^{(1)} &= s(p_0 \oplus k_0) \oplus 2 \bullet s(p_5 \oplus k_5) \oplus 3 \bullet s(p_{10} \oplus k_{10}) \\ &\oplus\ s(p_{15} \oplus k_{15}) \oplus s(k_{14}) \oplus k_1 \end{aligned} \tag{6}$$

$$\begin{aligned} x_2^{(1)} &= s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \\ &\oplus\ 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \end{aligned} \tag{7}$$

$$\begin{aligned} x_3^{(1)} &= 3 \bullet s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus s(p_{10} \oplus k_{10}) \\ &\oplus\ 2 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{12}) \oplus k_3 \end{aligned} \tag{8}$$

$$\begin{aligned} x_4^{(1)} &= 2 \bullet s(p_4 \oplus k_4) \oplus 3 \bullet s(p_9 \oplus k_9) \oplus s(p_{14} \oplus k_{14}) \\ &\oplus\ s(p_3 \oplus k_3) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus 1 \end{aligned} \tag{9}$$

$$\begin{aligned} x_5^{(1)} &= s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \\ &\oplus\ s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_5 \end{aligned} \tag{10}$$

$$\begin{aligned} x_6^{(1)} &= s(p_4 \oplus k_4) \oplus s(p_9 \oplus k_9) \oplus 2 \bullet s(p_{14} \oplus k_{14}) \\ &\oplus\ 3 \bullet s(p_3 \oplus k_3) \oplus s(k_{15}) \oplus k_2 \oplus k_6 \end{aligned} \tag{11}$$

$$\begin{aligned} x_7^{(1)} &= 3 \bullet s(p_4 \oplus k_4) \oplus s(p_9 \oplus k_9) \oplus s(p_{14} \oplus k_{14}) \\ &\oplus\ 2 \bullet s(p_3 \oplus k_3) \oplus s(k_{12}) \oplus k_3 \oplus k_7 \end{aligned} \tag{12}$$

$$\begin{aligned} x_8^{(1)} &= 2 \bullet s(p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \\ &\oplus\ s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \end{aligned} \tag{13}$$

$$\begin{aligned} x_9^{(1)} &= s(p_8 \oplus k_8) \oplus 2 \bullet s(p_{13} \oplus k_{13}) \oplus 3 \bullet s(p_2 \oplus k_2) \\ &\oplus\ s(p_7 \oplus k_7) \oplus s(k_{14}) \oplus k_1 \oplus k_5 \oplus k_9 \end{aligned} \tag{14}$$

$$\begin{aligned} x_{10}^{(1)} &= s(p_8 \oplus k_8) \oplus s(p_{13} \oplus k_{13}) \oplus 2 \bullet s(p_2 \oplus k_2) \\ &\oplus\ 3 \bullet s(p_7 \oplus k_7) \oplus s(k_{15}) \oplus k_2 \oplus k_6 \oplus k_{10} \end{aligned} \tag{15}$$

$$\begin{aligned} x_{11}^{(1)} &= 3 \bullet s(p_8 \oplus k_8) \oplus s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \\ &\oplus\ 2 \bullet s(p_7 \oplus k_7) \oplus s(k_{12}) \oplus k_3 \oplus k_7 \oplus k_{11} \end{aligned} \tag{16}$$

$$\begin{aligned} x_{12}^{(1)} &= 2 \bullet s(p_{12} \oplus k_{12}) \oplus 3 \bullet s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \\ &\oplus\ s(p_{11} \oplus k_{11}) \oplus s(k_{13}) \oplus k_{12} \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \end{aligned} \tag{17}$$

$$\begin{aligned} x_{13}^{(1)} &= s(p_{12} \oplus k_{12}) \oplus 2 \bullet s(p_1 \oplus k_1) \oplus 3 \bullet s(p_6 \oplus k_6) \\ &\oplus\ s(p_{11} \oplus k_{11}) \oplus s(k_{14}) \oplus k_{13} \oplus k_1 \oplus k_5 \oplus k_9 \end{aligned} \tag{18}$$

$$\begin{aligned} x_{14}^{(1)} &= s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus 2 \bullet s(p_6 \oplus k_6) \\ &\oplus\ 3 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{15}) \oplus k_{14} \oplus k_2 \oplus k_6 \oplus k_{10} \end{aligned} \tag{19}$$

$$\begin{aligned} x_{15}^{(1)} &= 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \\ &\oplus\ 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11} \end{aligned} \tag{20}$$

Given an equation and row of a table, we compute the high-order nibble of each term on the RHS of the equation and XOR them. Each high-order nibble is a function of the plaintext, high-order nibbles of the key (obtained in the First Round Attack) and values of attributes (subkeys) in the given row. The high-order nibble of the LHS of the equation is the specific cache line number accessed by the victim in Round 2 and is contained in the set of line numbers provided by the spy threads. So, the predicate is satisfied for a given row if the value of the high-order nibble of the byte value of the RHS is equal to any of the elements in the set of cache line numbers provided by the spy.

The Cartesian product ($\times$) and join ($\bowtie$), both operate on two relations. The $\times$ operation pairs each tuple in the first relation with every tuple in the second relation. The attributes of the output relation consist of all attributes of both relations while the cardinality of the output relation is the product of the cardinalities of the input relations.

One use of the Cartesian product in this paper is in expressions of the form $r \times \{0,1\}^4$. Here, each tuple of r is paired with each 4-bit string, 0000, 0001, . . . , 1111 in turn. The extra attribute of the output relation is, for example, a low-order nibble of one of the bytes of the AES key.

If A_1 and A_2 are respectively the attribute sets of r_1 and r_2, then the attribute set of $r_1 \bowtie r_2$ is $A_1 \cup A_2$. A tuple, t is in $r_1 \bowtie r_2$ if and only if $t \cdot A_1 \in r_1$ and $t \cdot A_2 \in r_2$.

We note that of the 16 equations, (7), (10), (13) and (20) are the only ones with least dependence on the low-order nibbles of the key. So, in Algorithm 2, we build four relations, each with four of their unknown key nibbles as attributes. The choice of these equations enables us to build relations with smaller cardinalities and at the same time deduce all 16 low-order nibbles of the AES key. The equations serve the role of selection predicates. After a sufficient number of select operations each relation survives with only a single subkey value. In the next subsection, we derive an expression for the number of block encryptions, $\varepsilon^{'}$, required to obtain the key.

Algorithm 2 Second Round Attack – Encryption (4 equations)

```
Input:
    $\varepsilon^{'}$ blocks of plaintext, $\rho_{0,t,e}$, $\rho_{1,t,e}$,
    $\rho_{2,t,e}$, $0 \le t \le 3$, $1 \le e \le \varepsilon^{'}$
Output:
    Low-order nibble of each byte of AES key

    Step 0: Create four relations
        $r_2\left(k_0^{''},\ k_5^{''},\ k_{10}^{''},\ k_{15}^{''}\right)$,
        $r_5\left(k_4^{''},\ k_9^{''},\ k_{14}^{''},\ k_3^{''}\right)$,
        $r_8\left(k_8^{''},\ k_{13}^{''},\ k_2^{''},\ k_7^{''}\right)$,
        $r_{15}\left(k_{12}^{''},\ k_1^{''},\ k_6^{''},\ k_{11}^{''}\right)$
    and initialize each of them to
        $\{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4$
    Step 1: Compute
        $r_i^{'} = \sigma_{m \cdot \varepsilon'}\left(\ldots\ \sigma_{m \cdot 2}\left(\sigma_{m \cdot 1}\left(r_i\right)\right)\ \ldots\right)$
        $m = i + 5$
          $i = 2,\ 5,\ 8,\ 15$
```

Algorithm 3 attempts to obtain the key with fewer encryptions using all 16 equations (5) – (20). The database schema for the initial, intermediate and final relations are shown in Table 2.

Algorithm 3 Second Round Attack – Encryption (16 equations)

```
Input:
    $\varepsilon$ blocks of plaintext, $\rho_{0,t,e}$, $\rho_{1,t,e}$,
    $\rho_{2,t,e}$, $0 \le t \le 3$, $1 \le e \le \varepsilon$
Output:
    Low-order nibble of each byte of AES key

    Steps 0 and 1 are as in Algorithm 2 above except that
    we require fewer encryptions $\varepsilon < \varepsilon^{'}$. The remaining
    steps are shown below
    Step 2: Compute
        $r_i = r_2^{'} \times \{0,1\}^4$, $i = 0, 1, 3$
        $r_i = r_5^{'} \times \{0,1\}^4$, $i = 4,\ 6,\ 7$
        $r_i = r_8^{'} \times \{0,1\}^4$, $i = 9,\ 10,\ 11$
        $r_i = r_{15}^{'} \times \{0,1\}^4$, $i = 12,\ 13,\ 14$
        $r_i^{'} = \sigma_{m \cdot \varepsilon}\left(\ldots \sigma_{m \cdot 2}\left(\sigma_{m \cdot 1}\left(r_i\right)\right)\ldots\right)$
        $m = i + 5$
          $i = 0,\ 1,\ 3,\ 4,\ 6,\ 7,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14$
    Step 3: Compute
        $r_{J0} = r_0^{'} \bowtie r_1^{'} \bowtie r_3^{'}$
        $r_{J1} = r_4^{'} \bowtie r_6^{'} \bowtie r_7^{'}$
        $r_{J2} = r_9^{'} \bowtie r_{10}^{'} \bowtie r_{11}^{'}$
        $r_{J3} = r_{12}^{'} \bowtie r_{13}^{'} \bowtie r_{14}^{'}$
    Step 4: Compute
        $r_{J01} = r_{J0} \bowtie r_{J1}$
        $r_{J23} = r_{J2} \bowtie r_{J3}$
    Step 5: Compute
        $r_{J0123} = r_{J01} \bowtie r_{J23}$
```

4.4. Second Round Attack - Results and Analysis

As in the First Round Attack, the number of block encryptions required to deduce the low-order nibbles of the key is closely related to the number of table accesses made in a run.

The attribute values of a tuple and the plaintext determine the RHS of an equation (one of (5) through (20)). Because of the non-linear nature of the S-Box function together with the assumption of random plaintext, it is reasonable to conclude that the high-order nibble of each term is uniformly distributed between 0 and 15. For this tuple to be retained, it has to match one of $|\rho_{0-1,*,*}|$ possible values of the high-order nibble of the LHS byte. Thus the compression ratio, c, of the output of the select operation (ratio of cardinalities of the output to input relation) is $c = |\rho_{0-1,*,*}|/16$. The cardinality of each output relation of Step 1 in Algorithm 3 is hence $2^{16} \times c^{\varepsilon}$. In Step 2, the input relations are first

TABLE 2: Database schema of initial, intermediate, and final relations for Algorithm 3

Relation	Attributes (low-order nibbles of AES key)
r_0	$k_0^{''}, k_5^{''}, k_{10}^{''}, k_{13}^{''}, k_{15}^{''}$
r_1	$k_0^{''}, k_5^{''}, k_{10}^{''}, k_{14}^{''}, k_{15}^{''}$
r_3	$k_0^{''}, k_5^{''}, k_{10}^{''}, k_{12}^{''}, k_{15}^{''}$
r_4	$k_3^{''}, k_4^{''}, k_9^{''}, k_{13}^{''}, k_{14}^{''}$
r_6	$k_3^{''}, k_4^{''}, k_9^{''}, k_{14}^{''}, k_{15}^{''}$
r_7	$k_3^{''}, k_4^{''}, k_9^{''}, k_{12}^{''}, k_{14}^{''}$
r_9	$k_2^{''}, k_7^{''}, k_8^{''}, k_{13}^{''}, k_{14}^{''}$
r_{10}	$k_2^{''}, k_7^{''}, k_8^{''}, k_{13}^{''}, k_{15}^{''}$
r_{11}	$k_2^{''}, k_7^{''}, k_8^{''}, k_{12}^{''}, k_{13}^{''}$
r_{12}	$k_1^{''}, k_6^{''}, k_{11}^{''}, k_{12}^{''}, k_{13}^{''}$
r_{13}	$k_1^{''}, k_6^{''}, k_{11}^{''}, k_{12}^{''}, k_{14}^{''}$
r_{14}	$k_1^{''}, k_6^{''}, k_{11}^{''}, k_{12}^{''}, k_{15}^{''}$
r_{J0}	$k_0^{''}, k_5^{''}, k_{10}^{''}, k_{12}^{''}, k_{13}^{''}, k_{14}^{''}, k_{15}^{''}$
r_{J1}	$k_3^{''}, k_4^{''}, k_9^{''}, k_{12}^{''}, k_{13}^{''}, k_{14}^{''}, k_{15}^{''}$
r_{J2}	$k_2^{''}, k_7^{''}, k_8^{''}, k_{12}^{''}, k_{13}^{''}, k_{14}^{''}, k_{15}^{''}$
r_{J3}	$k_1^{''}, k_6^{''}, k_{11}^{''}, k_{12}^{''}, k_{13}^{''}, k_{14}^{''}, k_{15}^{''}$
r_{J01}	$k_0^{''}, k_3^{''}, k_4^{''}, k_5^{''}, k_9^{''}, k_{10}^{''}, k_{12}^{''}, k_{13}^{''}, k_{14}^{''}, k_{15}^{''}$
r_{J23}	$k_1^{''}, k_2^{''}, k_6^{''}, k_7^{''}, k_8^{''}, k_{11}^{''}, k_{12}^{''}, k_{13}^{''}, k_{14}^{''}, k_{15}^{''}$
r_{J0123}	Low-order nibble of each of the 16 bytes of the AES key

expanded by a factor of 2^4 before repeatedly performing the select operation. Thus the cardinality of each output relation in Step 2 is $2^{20} \times c^{2\varepsilon}$.

Step 3 involves four 3-way joins. It is well-known in the Database Query Processing literature [10] that the cardinality of a 2-way join output is $\frac{|r_A| \times |r_B|}{|JA|}$ where $|r_A|$ and $|r_B|$ are the input cardinalities and $|JA|$ is the cardinality of the join attribute. Since the join attribute values were inherited from the output of Step 1, $|JA|$ in this case is $2^{16} \times c^{\varepsilon}$. The cardinalities of each intermediate and final join output in Step 3 are thus respectively $\frac{\left(2^{20} \times c^{2\varepsilon}\right)^2}{2^{16} \times c^{\varepsilon}} = 2^{24} \times c^{3\varepsilon}$ and $\frac{\left(2^{24} \times c^{3\varepsilon}\right)\left(2^{20} \times c^{2\varepsilon}\right)}{2^{16} \times c^{\varepsilon}} = 2^{28} \times c^{4\varepsilon}$.

In Step 4, the join attributes are $k_{12}^{''}$, $k_{13}^{''}$, $k_{14}^{''}$ and $k_{15}^{''}$. Their values in the two input relations are independent of each other and, collectively, take 2^{16} possible values. Hence the join outputs in Step 4 have cardinality $\frac{\left(2^{28} \times c^{4\varepsilon}\right)^2}{2^{16}} = 2^{40} \times c^{8\varepsilon}$. Finally, the cardinality of the output in Step 5 is $\frac{\left(2^{40} \times c^{8\varepsilon}\right)^2}{2^{16}} = 2^{64} \times c^{16\varepsilon}$. To estimate the number of encryptions required to deduce the low-order nibbles of the AES key we set the cardinality of the final output to 1 and solve for ε to obtain $\varepsilon = \frac{-4}{\log_2 c}$ where $c = \frac{|\rho_{0-1,*,*}|}{16}$.

Figure 4 compares the number of encryptions required to deduce the subkeys in Algorithms 2 and 3. The number of encryptions for Algorithm 2 is obtained by solving for ε in the equation $1 = 2^{16} \times c^{\varepsilon}$ resulting in $\varepsilon = \frac{-16}{\log_2 c}$. Algorithm 2 thus requires four times the number of encryptions compared to Algorithm 3. This is because the latter uses the information in all 16 of the second round access equations rather than in just the four equations used by Algorithm 2.

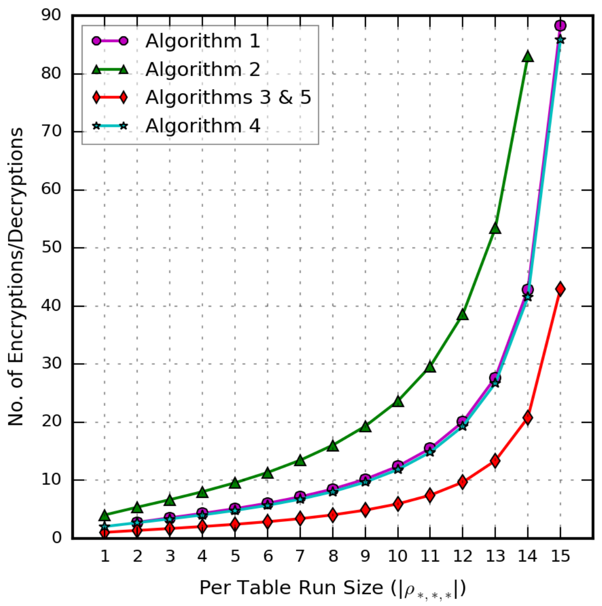

Figure 4: # of encryptions/decryptions required to retrieve the AES key with different algorithms as a function of Run Size.

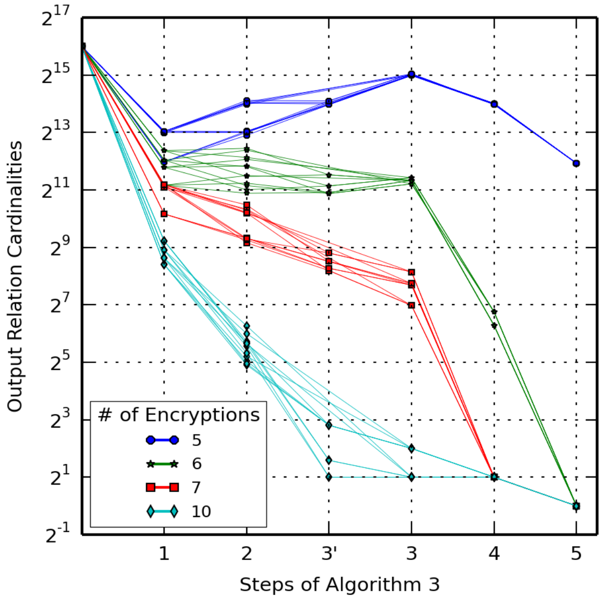

Figure 5: Cardinality of output relations after each step for varying number of encryptions

Figure 5 shows the effect of number of encryptions on the cardinalities of output relations of each step of Algorithm 3 with a randomly generated key and 10 random plaintext blocks. (The cardinalities are the number of potentially correct key or subkey values that survive as the algorithm

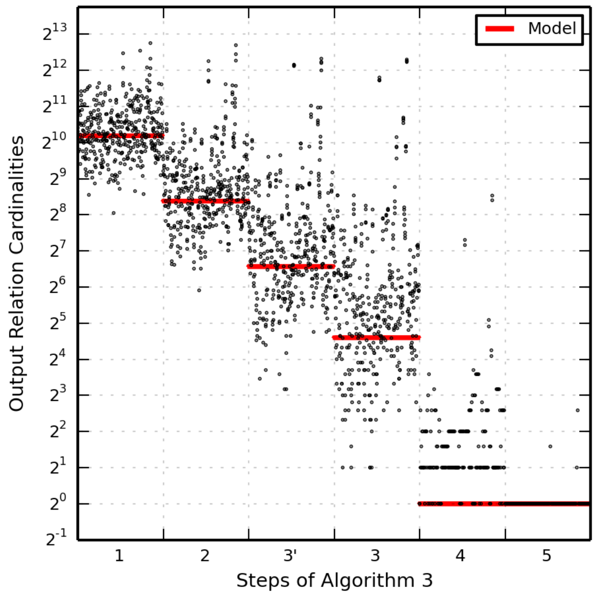

Figure 6: Output relation cardinality variations across 100 samples after each step of Algorithm 3 with 7 encryptions

progresses). The operations denoted $3'$ and 3 just below the x-axis in Figure 5 are the first and second joins of Step 3 of Algorithm 3. With five encryptions, the number of possible keys upon algorithm termination is $\sim 2^{12}$ because the size of the join outputs is very large. With 6 and 7 encryptions, the last two join output cardinalities are considerably reduced ultimately resulting in successful recovery of the AES key.

We randomly generated 100 128-bit keys and selected 30 random plaintexts per key. With Algorithm 3, 90%, 97% and 100% of the keys were uniquely retrieved using six, seven and eight encryptions respectively. The number of encryptions required by Algorithm 2 was around 25 as predicted by the model for an effective run size = 10.

Figure 6 shows the inter-sample variation in the output cardinalities of each step in Algorithm 3 with six encryptions. Each vertical strip (corresponding to a step in the Algorithm) contains the results with all 100 samples. The trend is toward lower cardinalities as the algorithm progresses though there are some conspicuous outliers. There is also a significant variation within each strip. In part, we attribute this to the variation in run size ($|\rho_{0-1,*,*}|$) across encryptions.

5. Scenario II Attack

Scenario II assumes knowledge of blocks of ciphertext and the corresponding cache lines accessed during decryption. As in Scenario I, there are two phases in this attack - the First Round attack (which reveals the high-order nibbles of each byte of the 10th round (encryption) key) and the Second Round attack which reveals the rest of the key.

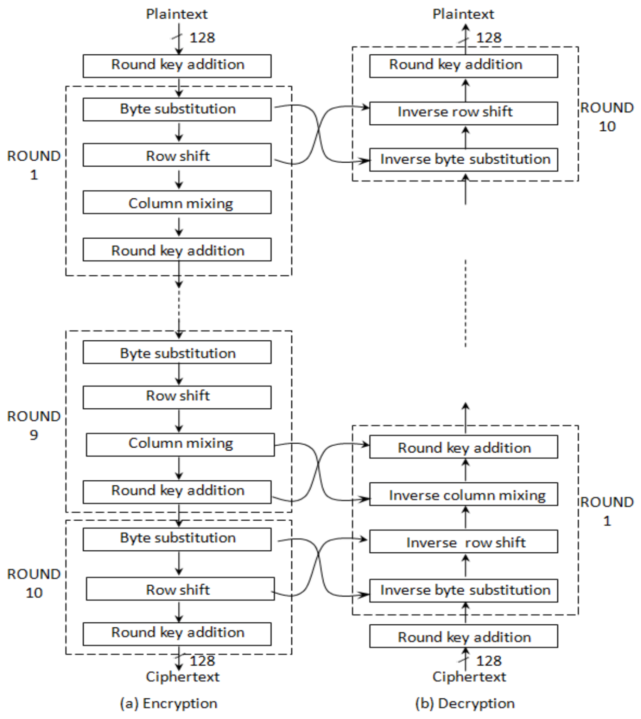

Figure 7: Order of steps in AES encryption/decryption

5.1. First Round Attack

Decryption involves reversing each step performed during encryption – right circular shift instead of left circular shift, use of inverse S-Box function (represented as s^{-1}) and B^{-1} in the column mixing step where

$$B^{-1} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$$

By interchanging the order of Byte Substitution and Shift Rows steps and also of the Column Mixing and Round Key operations (Figure 7), OpenSSL implements decryption using the same sequence of steps as encryption. Also, the order of round key usage in decryption is reversed (so the first key operation in decryption uses the 10th round key in encryption). Moreover, for correct operation, the key matrices used in each round should be multiplied by B^{-1}. Representing the ciphertext and the i^{th} round key (of encryption) as 4×4 matrices C and K_i respectively, the first step in decryption is to compute $C \oplus K_{10}$.

The Key Expansion Algorithm [3] is used to derive all the round keys from the initial AES key. However, this procedure is reversible – knowing the 10th round key, we may derive all the other round keys and the AES key. Recovery of the high-order nibble of each byte of the 10th round key is very similar to the First Round attack on encryption. Here, the sequence of table accesses in the first round is to line numbers $c_0' \oplus k_0'$, $c_{10}' \oplus k_{10}'$, $c_{13}' \oplus k_{13}'$, etc.

Thus, knowledge of the table line numbers and the ciphertext enables us to deduce the high-order nibble of each byte of K_{10}.

5.2. Algorithm 4 - Description (Second Round Attack)

The four steps of the first round of decryption transform the ciphertext as explained in subsection 5.1. From this, we derive expressions for the table indices accessed in the second round in terms of C and K_{10} ((21) – (36)). Note that in these equations, the key variables refer to the bytes of the 10^{th} round key used in encryption.

$$\begin{aligned} x_0^{(1)} =& e \bullet s^{-1}(c_0 \oplus k_0) \oplus b \bullet s^{-1}(c_{13} \oplus k_{13}) \\ &\oplus d \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus 9 \bullet s^{-1}(c_7 \oplus k_7) \\ &\oplus e \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \oplus b \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ &\oplus d \bullet (k_2 \oplus s(k_{11} \oplus k_{15})) \oplus 9 \bullet (k_3 \oplus s(k_8 \oplus k_{12})) \end{aligned} \tag{21}$$

$$\begin{aligned} x_1^{(1)} =& 9 \bullet s^{-1}(c_0 \oplus k_0) \oplus e \bullet s^{-1}(c_{13} \oplus k_{13}) \\ &\oplus b \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus d \bullet s^{-1}(c_7 \oplus k_7) \\ &\oplus 9 \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \oplus e \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ &\oplus b \bullet (k_2 \oplus s(k_{11} \oplus k_{15}) \oplus d \bullet (k_3 \oplus s(k_8 \oplus k_{12})) \end{aligned} \tag{22}$$

$$\begin{aligned} x_2^{(1)} =& d \bullet s^{-1}(c_0 \oplus k_0) \oplus 9 \bullet s^{-1}(c_{13} \oplus k_{13}) \\ &\oplus e \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus b \bullet s^{-1}(c_7 \oplus k_7) \\ &\oplus d \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \oplus 9 \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ &\oplus e \bullet (k_2 \oplus s(k_{11} \oplus k_{15}) \oplus b \bullet (k_3 \oplus s(k_8 \oplus k_{12})) \end{aligned} \tag{23}$$

$$\begin{aligned} x_3^{(1)} =& b \bullet s^{-1}(c_0 \oplus k_0) \oplus d \bullet s^{-1}(c_{13} \oplus k_{13}) \\ &\oplus 9 \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus e \bullet s^{-1}(c_7 \oplus k_7) \\ &\oplus b \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \oplus d \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ &\oplus 9 \bullet (k_2 \oplus s(k_{11} \oplus k_{15})) \oplus e \bullet (k_3 \oplus s(k_8 \oplus k_{12})) \end{aligned} \tag{24}$$

$$\begin{aligned} x_4^{(1)} =& e \bullet s^{-1}(c_4 \oplus k_4) \oplus b \bullet s^{-1}(c_1 \oplus k_1) \\ &\oplus d \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 9 \bullet s^{-1}(c_{11} \oplus k_{11}) \\ &\oplus e \bullet (k_0 \oplus k_4) \oplus b \bullet (k_1 \oplus k_5) \oplus d \bullet (k_2 \oplus k_6) \\ &\oplus 9 \bullet (k_3 \oplus k_7) \end{aligned} \tag{25}$$

$$\begin{aligned} x_5^{(1)} =& 9 \bullet s^{-1}(c_4 \oplus k_4) \oplus e \bullet s^{-1}(c_1 \oplus k_1) \\ &\oplus b \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus d \bullet s^{-1}(c_{11} \oplus k_{11}) \\ &\oplus 9 \bullet (k_0 \oplus k_4) \oplus e \bullet (k_1 \oplus k_5) \oplus b \bullet (k_2 \oplus k_6) \\ &\oplus d \bullet (k_3 \oplus k_7) \end{aligned} \tag{26}$$

$$\begin{aligned} x_6^{(1)} =& d \bullet s^{-1}(c_4 \oplus k_4) \oplus 9 \bullet s^{-1}(c_1 \oplus k_1) \\ &\oplus e \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus b \bullet s^{-1}(c_{11} \oplus k_{11}) \\ &\oplus d \bullet (k_0 \oplus k_4) \oplus 9 \bullet (k_1 \oplus k_5) \oplus e \bullet (k_2 \oplus k_6) \\ &\oplus b \bullet (k_3 \oplus k_7) \end{aligned} \tag{27}$$

$$\begin{aligned} x_7^{(1)} =& b \bullet s^{-1}(c_4 \oplus k_4) \oplus d \bullet s^{-1}(c_1 \oplus k_1) \\ &\oplus 9 \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus e \bullet s^{-1}(c_{11} \oplus k_{11}) \\ &\oplus b \bullet (k_0 \oplus k_4) \oplus d \bullet (k_1 \oplus k_5) \oplus 9 \bullet (k_2 \oplus k_6) \\ &\oplus e \bullet (k_3 \oplus k_7) \end{aligned} \tag{28}$$

$$\begin{aligned} x_8^{(1)} =& e \bullet s^{-1}(c_8 \oplus k_8) \oplus b \bullet s^{-1}(c_5 \oplus k_5) \\ &\oplus d \bullet s^{-1}(c_2 \oplus k_2) \oplus 9 \bullet s^{-1}(c_{15} \oplus k_{15}) \\ &\oplus e \bullet (k_4 \oplus k_8) \oplus b \bullet (k_5 \oplus k_9) \oplus d \bullet (k_6 \oplus k_{10}) \\ &\oplus 9 \bullet (k_7 \oplus k_{11}) \end{aligned} \tag{29}$$

$$\begin{aligned} x_9^{(1)} =& 9 \bullet s^{-1}(c_8 \oplus k_8) \oplus e \bullet s^{-1}(c_5 \oplus k_5) \\ &\oplus b \bullet s^{-1}(c_2 \oplus k_2) \oplus d \bullet s^{-1}(c_{15} \oplus k_{15}) \\ &\oplus 9 \bullet (k_4 \oplus k_8) \oplus e \bullet (k_5 \oplus k_9) \oplus b \bullet (k_6 \oplus k_{10}) \\ &\oplus d \bullet (k_7 \oplus k_{11}) \end{aligned} \tag{30}$$

$$\begin{aligned} x_{10}^{(1)} =& d \bullet s^{-1}(c_8 \oplus k_8) \oplus 9 \bullet s^{-1}(c_5 \oplus k_5) \\ &\oplus e \bullet s^{-1}(c_2 \oplus k_2) \oplus b \bullet s^{-1}(c_{15} \oplus k_{15}) \\ &\oplus d \bullet (k_4 \oplus k_8) \oplus 9 \bullet (k_5 \oplus k_9) \oplus e \bullet (k_6 \oplus k_{10}) \\ &\oplus b \bullet (k_7 \oplus k_{11}) \end{aligned} \tag{31}$$

$$\begin{aligned} x_{11}^{(1)} =& b \bullet s^{-1}(c_8 \oplus k_8) \oplus d \bullet s^{-1}(c_5 \oplus k_5) \\ &\oplus 9 \bullet s^{-1}(c_2 \oplus k_2) \oplus e \bullet s^{-1}(c_{15} \oplus k_{15}) \\ &\oplus b \bullet (k_4 \oplus k_8) \oplus d \bullet (k_5 \oplus k_9) \oplus 9 \bullet (k_6 \oplus k_{10}) \\ &\oplus e \bullet (k_7 \oplus k_{11}) \end{aligned} \tag{32}$$

$$\begin{aligned} x_{12}^{(1)} =& e \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus b \bullet s^{-1}(c_9 \oplus k_9) \\ &\oplus d \bullet s^{-1}(c_6 \oplus k_6) \oplus 9 \bullet s^{-1}(c_3 \oplus k_3) \\ &\oplus e \bullet (k_8 \oplus k_{12}) \oplus b \bullet (k_9 \oplus k_{13}) \oplus d \bullet (k_{10} \oplus k_{14}) \\ &\oplus 9 \bullet (k_{11} \oplus k_{15}) \end{aligned} \tag{33}$$

$$\begin{aligned} x_{13}^{(1)} =& 9 \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus e \bullet s^{-1}(c_9 \oplus k_9) \\ &\oplus b \bullet s^{-1}(c_6 \oplus k_6) \oplus d \bullet s^{-1}(c_3 \oplus k_3) \\ &\oplus 9 \bullet (k_8 \oplus k_{12}) \oplus e \bullet (k_9 \oplus k_{13}) \oplus b \bullet (k_{10} \oplus k_{14}) \\ &\oplus d \bullet (k_{11} \oplus k_{15}) \end{aligned} \tag{34}$$

$$\begin{aligned} x_{14}^{(1)} =& d \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus 9 \bullet s^{-1}(c_9 \oplus k_9) \\ &\oplus e \bullet s^{-1}(c_6 \oplus k_6) \oplus b \bullet s^{-1}(c_3 \oplus k_3) \\ &\oplus d \bullet (k_8 \oplus k_{12}) \oplus 9 \bullet (k_9 \oplus k_{13}) \oplus e \bullet (k_{10} \oplus k_{14}) \\ &\oplus b \bullet (k_{11} \oplus k_{15}) \end{aligned} \tag{35}$$

$$\begin{aligned} x_{15}^{(1)} =& b \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus d \bullet s^{-1}(c_9 \oplus k_9) \\ &\oplus 9 \bullet s^{-1}(c_6 \oplus k_6) \oplus e \bullet s^{-1}(c_3 \oplus k_3) \\ &\oplus b \bullet (k_8 \oplus k_{12}) \oplus d \bullet (k_9 \oplus k_{13}) \oplus 9 \bullet (k_{10} \oplus k_{14}) \\ &\oplus e \bullet (k_{11} \oplus k_{15}) \end{aligned} \tag{36}$$

As in the case of encryption (Section 4), we identify all unknowns in (21) – (36) that affect the high-order nibble of the LHS. These are the low-order nibbles of different key bytes and range in number from 13 nibbles (in (21) – (24)) to 10 nibbles (in (25) – (36)). Processing relations with such a large number of attribute values is not practical. Instead, we replace all ciphertext-independent terms on the RHS by a single byte variable - one variable per equation. For example, (25) now becomes

$$\begin{aligned} x_4^{(1)} =& e \bullet s^{-1}(c_4 \oplus k_4) \oplus b \bullet s^{-1}(c_1 \oplus k_1) \\ &\oplus d \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 9 \bullet s^{-1}(c_{11} \oplus k_{11}) \oplus y_4 \end{aligned} \tag{37}$$

Effectively, the numbers of unknowns (low-order nibbles of the key) are reduced to five resulting in smaller relation cardinalities.

Algorithm 4 takes as input δ blocks of ciphertext and the sets of table line numbers and computes the low-order nibbles of the 10^{th} round key.

The tables, r_{JF0}, r_{JF4}, r_{JF8} and r_{JF12} contain the final values of the subkeys $k_0^{''}\,k_{13}^{''}\,k_{10}^{''}\,k_7^{''}$, $k_4^{''}\,k_1^{''}\,k_{14}^{''}\,k_{11}^{''}$, $k_8^{''}\,k_5^{''}\,k_2^{''}\,k_{15}^{''}$ and $k_{12}^{''}\,k_9^{''}\,k_6^{''}\,k_3^{''}$ respectively. We next estimate the number of decryptions, δ, required to obtain the complete key, K_{10}.

5.3. Algorithm 4 - Results and Analysis

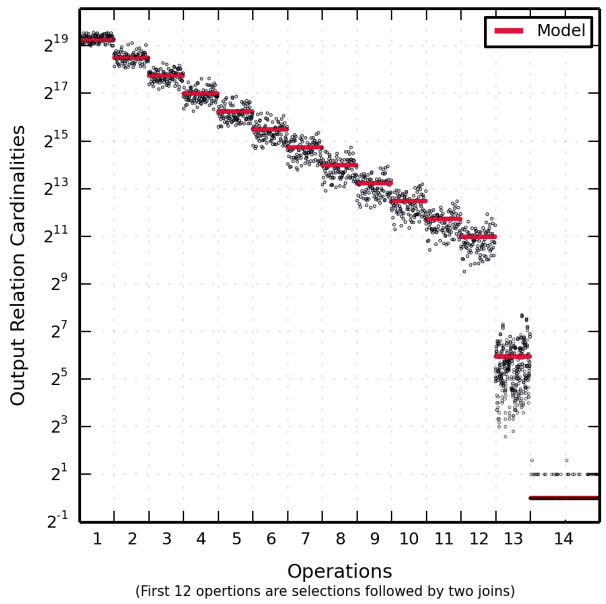

Figure 8: Output relation cardinality variations after each select and join operation of Algorithm 4

The 16 relations created in Step 0 each have cardinality 2^{20}. After the δ select operations in Step 1, the cardinality of each relation is roughly $2^{20} \times c^{\delta}$. The cardinalities of the eight relations created at the end of Step 2 and the four relations created at the end of Step 3 are approximately $\frac{(2^{20}\times c^{\delta})^2}{2^{16}} = 2^{24} \times c^{2\delta}$ and $\frac{(2^{24}\times c^{2\delta})^2}{2^{16}} = 2^{32} \times c^{4\delta}$ respectively. To obtain an estimate of the number of decryptions necessary to retrieve the key, we solve for δ by setting the cardinality of the output of the final join to 1. This yields $\delta = \frac{-8}{\log_2 c}$. As shown in Figure 4, the number of decryptions is between the number of encryptions required in Algorithm 2 and Algorithm 3.

As in the case with encryption, we generated 100 random keys and 30 ciphertext blocks per key. For each key, we decrypted 12 random blocks of ciphertext. Figure 8 is a plot of the cardinality of the output relation after each of the 12 select operations for each of the 100 samples. As expected, the cardinalities of successive outputs decrease geometrically. There is also a substantial drop in the size of the relations after the two joins in steps 2 and 3. Also shown are the results of the analytical model (the horizontal segments) superimposed against the experimental results.

```
Algorithm 4 Second Round Attack- Decryption
Input:
    δ blocks of ciphertext, $\rho_{0,t,d}$, $\rho_{1,t,d}$,
    $\rho_{2,t,d}$, $0 \le t \le 3$, $1 \le d \le \delta$
Output:
    Low-order nibble of each byte of the first round key in
    decryption (tenth round key in encryption)
    Step 0: Create 16 relations
        $r_i\left(k_0^{''},\, k_{13}^{''},\, k_{10}^{''},\, k_7^{''},\, y_i^{'}\right)$, $0 \le \mathrm{i} \le 3$
        $r_i\left(k_4^{''},\, k_1^{''},\, k_{14}^{''},\, k_{11}^{''},\, y_i^{'}\right)$, $4 \le \mathrm{i} \le 7$
        $r_i\left(k_8^{''},\, k_5^{''},\, k_2^{''},\, k_{15}^{''},\, y_i^{'}\right)$, $8 \le \mathrm{i} \le 11$
        $r_i\left(k_{12}^{''},\, k_9^{''},\, k_6^{''},\, k_3^{''},\, y_i^{'}\right)$, $12 \le \mathrm{i} \le 15$
    Initialize each to
        $\{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4$
    Step 1:
        $r_i^{'} = \sigma_{m.\delta}\left(\ldots\sigma_{m.2}\left(\sigma_{m.1}\left(r_i\right)\right)\ldots\right)$
        $m = i + 21$
        $i = 0, 1, \ldots, 15$
    Step 2:
        $r_{Ji} = r_i^{'} \bowtie r_{i+1}^{'}$,  $i = 0, 2, 4 \ldots 14$
    Step 3:
        $r_{JFi} = r_{Ji} \bowtie r_{J(i+2)}$,  $i = 0, 4, 8, 12$
```

TABLE 3: Distribution of # of candidate keys returned by Algorithm 4 as a function of # of decryptions

# of	# of Candidate Key Values						
Decryptions	1	1-10	10-10^2	10^2-10^3	10^3-10^4	10^4-10^5	10^5-10^6
8	0	0	0	0	0	0	100
9	0	0	0	0	0	5	95
10	0	0	8	17	15	14	46
11	4	30	26	15	17	6	2
12	36	49	12	2	1	0	0
..	..	..	..	..	..	..	..
17	99	1	0	0	0	0	0
18	100	0	0	0	0	0	0

Table 3 depicts the degree of success of Algorithm 4 in narrowing down the set of potential keys as a function of the number of decryptions. With 18 or more ciphertext blocks provided for decryption, the AES key was uniquely identified in every sample. With 17 ciphertext blocks, Algorithm

4 discovered the unique key in 99 samples and reported 4 possible keys in the remaining case. With 12 decryptions, each of 85 out of 100 samples returned up to 10 possible candidate keys and another 12 samples each returned up to 100 candidate keys. Attempts at learning the key with fewer than 10 decryptions returned a very large number of candidate keys and so could be considered unsuccessful.

5.4. Algorithm 5 - Theoretical Underpinnings

We next present an algorithm that considerably improves upon Algorithm 4. For ease of explanation, we refer to (21)-(24) as Set-1 equations, (25)-(28) as Set-2, (29)-(32) as Set-3 and (33)-(36) as Set-4 equations. We handle the Set-1 equations (21)-(24) as in Algorithm 4. The remaining 12 equations are treated differently as explained below. Because field multiplication is distributive over field addition, it is possible to split each of the last four terms on the RHS of those equations. Upon rearranging terms, (26), for example, can be re-written as

$$\begin{aligned}
& x_4^{(1)} \oplus 9 \bullet s^{-1}(c_4 \oplus k_4) \oplus e \bullet s^{-1}(c_1 \oplus k_1) \\
&\quad \oplus b \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus d \bullet s^{-1}(c_{11} \oplus k_{11}) \\
&\quad \oplus 9 \bullet ((k_0 \oplus k_4)^{'} 0000) \oplus e \bullet ((k_1 \oplus k_5)^{'} 0000) \\
&\quad \oplus b \bullet ((k_2 \oplus k_6)^{'} 0000) \oplus d \bullet ((k_3 \oplus k_7)^{'} 0000) \\
&\quad = \\
&\qquad 9 \bullet (0000\, (k_0 \oplus k_4)^{''}) \oplus e \bullet (0000\, (k_1 \oplus k_5)^{''}) \\
&\quad \oplus b \bullet (0000\, (k_2 \oplus k_6)^{''}) \oplus d \bullet (0000\, (k_3 \oplus k_7)^{''})
\end{aligned} \tag{38}$$

Let the RHS of (38) be equal to the byte denoted $(x_0\, x_1\, x_2\, x_3\;\; x_4\, x_5\, x_6\, x_7)$. Also, let $a_0a_1a_2$, $a_3a_4a_5$, $a_6a_7a_8$ and $a_9a_{10}a_{11}$ denote the most significant three bits of the nibbles $(k_0 \oplus k_4)^{''}, (k_1 \oplus k_5)^{''}, (k_2 \oplus k_6)^{''}$ and $(k_3 \oplus k_7)^{''}$ respectively.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	
_	a_0	a_1	a_2	z	_	_	_	9 (1001)
_	_	_	_	a_0	a_1	a_2	z	
_	a_3	a_4	a_5	z	_	_	_	
_	_	a_3	a_4	a_5	z	_	_	e (1110)
_	_	_	a_3	a_4	a_5	z	_	
_	a_6	a_7	a_8	z	_	_	_	
_	_	_	a_6	a_7	a_8	z	_	b (1011)
_	_	_	_	a_6	a_7	a_8	z	
_	a_9	a_{10}	a_{11}	z	_	_	_	
_	_	a_9	a_{10}	a_{11}	z	_	_	d (1101)
_	_	_	_	a_9	a_{10}	a_{11}	z	

Figure 9: Field Multiplications involved in RHS of (38)

Multiplication in a binary field involves shifting and XORing of the multiplicand. Figure 9 shows these operations on RHS terms of (38). The high-order nibble (x_0, x_1, x_2, x_3) of the sum of the terms on the RHS is

$$\begin{aligned}
x_0 &= 0 \\
x_1 &= a_0 \oplus a_3 \oplus a_6 \oplus a_9 \\
x_2 &= a_1 \oplus a_3 \oplus a_4 \oplus a_7 \oplus a_9 \oplus a_{10} \\
x_3 &= a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_8 \oplus a_{10} \oplus a_{11}
\end{aligned}$$

From Figure 9, it is evident that the bits denoted z in the RHS of equation (38) do not affect the high-order nibble of the resultant byte. It follows that, of the 16 unknown bits on the RHS of (38), only 12 bits determine the high-order nibble on the RHS.

The operations involved in calculating x_1, x_2 and x_3 can be represented using the matrix equation

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = M_{9ebd} \bullet A \tag{39}$$

where

$$M_{9ebd} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

and

$$A = (a_0\;\, a_1\;\, a_2\;\, a_3\;\, a_4\;\, a_5\;\, a_6\;\, a_7\;\, a_8\;\, a_9\;\, a_{10}\;\, a_{11})^T$$

The subscript '9ebd' reflects the order of co-efficients in RHS of (38).
Let $\mathbb{F}_2^{12}$ represent the 12-dimensional binary vector space. We define 8 equivalence classes as follows

$$C_{9ebd}\left((x_1, x_2, x_3)^T\right) = \left\{ A \in \mathbb{F}_2^{12} : M_{9ebd} \bullet A = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \right\} \tag{40}$$

M_{9ebd} is in row canonical form with pivots in the first 3 columns. So a_0, a_1, a_2 are pivot variables and the remaining nine are free variables in A. So, each equivalence class has $2^9 = 512$ sub-keys with class representative $(x_1x_2x_3\, 000\, 000\, 000)$. It follows that

Theorem 1: $\mathbb{F}_2^{12}$ can be partitioned into 8 equivalence classes based on (40) each containing 512 sub-keys. The class representatives are $(\{0,1\}^3\; 000\, 000\, 000)$.

The above theorem implies that instead of validating all the 2^{12} sub-keys, it is sufficient to validate only the class representatives.

The coefficients of the ciphertext-independent terms on the RHS of (25), (27) and (28) are shifted versions of those in (26). Analogous to M_{9ebd}, we define M_{ebd9}, M_{bd9e} and M_{d9eb}. These matrices are obtained in a manner similar to M_{9ebd} (Figure 9) and are column-shifted versions of M_{9ebd}. Moreover, they are linearly related as follows

$$M_{ebd9} = M_1 \bullet M_{9ebd} \quad (41)$$

$$M_{d9eb} = M_2 \bullet M_{9ebd} \quad (42)$$

$$M_{bd9e} = M_3 \bullet M_{9ebd} \quad (43)$$

where

$$M_1 = \begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 1 & 1 & 1\end{pmatrix}, M_2 = \begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\end{pmatrix}, M_3 = \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 0 & 1\end{pmatrix}$$

Let $A_1 \in C_{9ebd}\left((x_1, x_2, x_3)^T\right)$. From (40)

$$M_{9ebd} \bullet A_1 = (x_1, x_2, x_3)^T$$

Pre-multiplying by M_1 on both sides

$$M_{ebd9} \bullet A_1 = M_1 \bullet (x_1, x_2, x_3)^T$$

So, $A_1 \in C_{ebd9}\left(M_1 \bullet (x_1, x_2, x_3)^T\right)$. This leads to the following theorem.

Theorem 2: If $A_1 \in C_{9ebd}\left((x_1, x_2, x_3)^T\right)$, then

$$A_1 \in C_{ebd9}\left(M_1 \bullet (x_1, x_2, x_3)^T\right),$$
$$A_1 \in C_{bd9e}\left(M_2 \bullet (x_1, x_2, x_3)^T\right) \quad \text{and}$$
$$A_1 \in C_{d9eb}\left(M_3 \bullet (x_1, x_2, x_3)^T\right)$$

The following is of crucial importance in the design of Algorithm 5.

Corollary 2.1: If A_1, A_2 belong to the same equivalence class w.r.t. M_{9ebd}, then they also belong to a single equivalence class w.r.t. M_{ebd9} or M_{bd9e} or M_{d9eb}.

Thus, M_{9ebd}, M_{ebd9}, M_{bd9e} and M_{d9eb}, each induce an identical partitioning over $\mathbb{F}_2^{12}$.

5.5. Algorithm 5 - Description

The Set-1 equations have S-box operations in ciphertext independent terms on the RHS. This makes it impossible to separate those terms into two parts as in (38). Instead, for each equation in Set-1, a different 4-bit variable is used to represent the sum of the high-order nibbles of ciphertext-independent terms. These are denoted $y_0 - y_3$ in Algorithm 5.

From Theorem 1, it is sufficient to validate only a representative of each class. Moreover, from Corollary 2.1, these classes and their representatives are identical across the four equations of each set (Set-2, Set-3 and Set-4). Since, by Theorem 1, class representatives are distinguished only by the first 3 bits (while the remaining nine bits are 0), we include a 3-bit attribute, y_4 (respectively y_8 and y_{12}) in r_4 (respectively r_8 and r_{12}). In Step 2, $s_{m\cdot x}(r_i)$ and $r_i^{'}$ contain tuples that satisfy all equations of a set for the x^{th}

Algorithm 5 Second Round Attack- Decryption

```
Input:
    δ blocks of ciphertext, $\rho_{0,t,d}$, $\rho_{1,t,d}$,
    $\rho_{2,t,d}$, $0 \le t \le 3$, $1 \le d \le \delta$
Output:
    Low-order nibble of each byte of the first round key in
    decryption (tenth round key in encryption)
    Step 0: Create 7 relations
        $r_i\left(k_0^{''},\ k_{13}^{''},\ k_{10}^{''},\ k_7^{''},\ y_i\right),\ i = 0, 1, 2, 3$
    Initialize each to
        $\{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4$
        $r_4\left(k_4^{''},\ k_1^{''},\ k_{14}^{''},\ k_{11}^{''},\ y_4\right)$
        $r_8\left(k_8^{''},\ k_5^{''},\ k_2^{''},\ k_{15}^{''},\ y_8\right)$
        $r_{12}\left(k_{12}^{''},\ k_9^{''},\ k_6^{''},\ k_3^{''},\ y_{12}\right)$
    Initialize each to
        $\{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^4 \times \{0,1\}^3$
    Step 1:
        $r_i^{'} = \sigma_{k.\delta}\left(\ldots\sigma_{k.2}\left(\sigma_{k.1}\left(r_i\right)\right)\right)$
        $k = i + 21$
        $i = 0, 1, 2, 3$
    Step 2:
        $r_{JFi} = s_{m\cdot\delta}\left(\ldots\left(s_{m\cdot 2}\left(s_{m\cdot 1}\left(r_i\right)\right)\right)\right)$
    where
        $s_{m\cdot x}\left(r_i\right) = \sigma_{(m+3)\cdot x}\left(\sigma_{(m+2)\cdot x}\left(\sigma_{(m+1)\cdot x}\left(\sigma_{(m)\cdot x}\left(r_i\right)\right)\right)\right)$
        $m = i + 21$
        $i = 4, 8, 12$
    Step 3:
        $r_{Ji} = r_i^{'} \bowtie r_{i+1}^{'},\ \ i = 0, 2$
        $r_{JF0} = r_{J0} \bowtie r_{J2}$
    Step 4:
        $r_c(k_0^{''}, k_1^{''}, \ldots, k_{15}^{''}, y_0\, y_1, y_2, y_3, y_4, y_8, y_{12})$
            $= r_{JF0} \times r_{JF4} \times r_{JF8} \times r_{JF12}$
    Step 5:
        $r_F^{'} = \sigma_{s_3}\left(\sigma_{s_2}\left(\sigma_{s_1}\left(\sigma_{s_0}\left(r_c\right)\right)\right)\right)$
        $r_F = \sigma_{s_{12}}\left(\sigma_{s_8}\left(\sigma_{s_4}\left(r_F^{'}\right)\right)\right)$
    $\sigma_{s_i}$ is explained in Section 5.5
```

TABLE 4: Database schema of initial, intermediate, and final relations

Relation	Attributes (Key nibbles)
r_0	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_0$
r_1	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_1$
r_2	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_2$
r_3	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_3$
r_{J0}	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_0, y_1$
r_{J2}	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_2, y_3$
r_{JF0}	$k_0^{''}, k_7^{''}, k_{10}^{''}, k_{13}^{''}, y_0, y_1, y_2, y_3$
r_4, r_{JF4}	$k_1^{''}, k_4^{''}, k_{11}^{''}, k_{14}^{''}, y_4$
r_8, r_{JF8}	$k_2^{''}, k_5^{''}, k_8^{''}, k_{15}^{''}, y_8$
r_{12}, r_{JF12}	$k_3^{''}, k_6^{''}, k_9^{''}, k_{12}^{''}, y_{12}$
r_c, r_F	Low-order nibble of each of the 16 bytes of the AES key and $y_0, y_1, y_2, y_3, y_4, y_8, y_{12}$

decryption and for all decryptions respectively. At the end of Step 3, r_{JF0} contains values of $k_0^{''}, k_7^{''}, k_{10}^{''}$ and $k_{13}^{''}$ that satisfy all Set-1 equations. r_c, the output of Step 4 contains tuples with potentially correct values of all 16 low-order nibbles of the AES key.

The first statement in Step 5 contains predicates denoted s_0, s_1, s_2 and s_3. These check for equality of two nibbles. For example, to check for s_0 in a tuple, the sum of the ciphertext-independent terms on the RHS of (21) is computed. This employs as input appropriate attribute values in the tuple (low-order nibbles of the key) together with high-order nibbles obtained from the First Round Attack. The high-order nibble of the result is compared with the value of y_0 in the tuple. Likewise, s_1, s_2 and s_3 are computed by considering (22), (23) and (24) and the values of y_1, y_2 and y_3. A given tuple is discarded if it fails to satisfy even a single predicate.

A tuple that satisfies the above predicates is then subjected to three more tests via predicates denoted s_4, s_8 and s_{12} (the second statement of Step 5). Each predicate tests for the equivalence of two vectors A_1 and $A_2 \in \mathbb{F}_2^{12}$ as defined in (40). For example, s_4 populates A_1 from the three most significant bits of $(k_0 \oplus k_4)^{''}, (k_1 \oplus k_5)^{''}, (k_2 \oplus k_6)^{''}$ and $(k_3 \oplus k_7)^{''}$ in that order (see (38)). The latter are obtained from the attribute values of the tuple under consideration. The first three bits of vector A_2 are copied from y_4 while the remaining bits of A_2 are made zero. From Corollary 2.1, the equivalence of A_1 and A_2 with respect to M_{9ebd} also guarantees their equivalence with respect to M_{ebd9}, M_{d9eb} and M_{bd9e}. So, a single predicate test for s_4 suffices (instead of 4 tests involving each equation in Set-2). Testing of predicates s_8 and s_{12} is similar but they involve an equation from Set-3 and Set-4 respectively.

5.6. Algorithm 5 - Analysis and Results

In Step 0, a relation is created for each of the four equations in Set-1 with cardinality 2^{20} each. In Step 1, the δ select operations reduce the size of each of r_0, r_1, r_2 and r_3 from 2^{20} to $2^{20} \times c^{\delta}$. In Step 2, the size of each of the input relations, r_4, r_8 and r_{12} is 2^{19}. After a total of 4δ select operations (4 per decryption), the size of each output relation is $2^{19} \times c^{4\delta}$. In Step 3, the cardinalities of r_{J0} and r_{J2} are $\frac{(2^{20} \times c^{\delta})^2}{(2^{16})} = 2^{24} \times c^{2\delta}$ while the cardinality of r_{JF0} is $\frac{(2^{24} \times c^{2\delta})^2}{(2^{16})} = 2^{32} \times c^{4\delta}$. The Cartesian product in Step 4 results in an output, r_c of size $(2^{32} \times c^{4\delta}) \cdot (2^{19} \times c^{4\delta})^3 = 2^{89} \times c^{16\delta}$.

Consider, for example, an average per table run size = 8, ($c = 0.5$). After 4 decryptions, at the end of Step 3, the cardinality of r_{JF0} will be 2^{16} and cardinality of r_{JF4}, r_{JF8} and r_{JF12} will be 2^3 each. The cardinality of r_c in Step 4 will be roughly $2^{16} \cdot 2^{3*3} = 2^{25}$ - a manageable size. Without using the theory developed in Section 5.4, the cardinality of the cartesian product would be $(2^{16})^4$.

Because y_0, y_1, y_2 and y_3 are each 4 bits and y_4, y_8 and y_{12} are 3 bits, the probability of a tuple surviving in Step 5 is $\frac{1}{(2^4)^4 \times (2^3)^3}$. So, the cardinality of r_F is $\frac{2^{89} \times c^{16\delta}}{(2^4)^4 \times (2^3)^3} = 2^{64} \times c^{16\delta}$. To obtain an estimate of the number of decryptions necessary to retrieve the key, we solve for δ by setting the cardinality of r_F to 1. This yield $\delta = \frac{-4}{\log_2 c}$

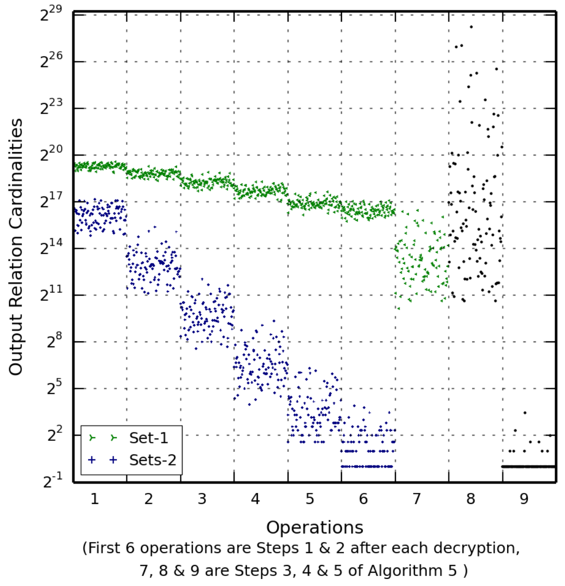

Figure 10: Output relation cardinality variations after each select and join operation of Algorithm 5

As in the case with the previous algorithm, we generated 100 random keys and 30 ciphertext blocks per key. For each sample, we decrypted 6 random blocks of ciphertext. Figure 10 is a plot of the relation cardinalities of the output relations after performing the select operations involving input from the six decryptions (samples). Cardinality of relations resulting from selection operations using Set-1 equations as predicates has a more gentle fall compared to those using Sets 2, 3 and 4. This is because r_4, r_8 and r_{12} are each subject to four select operations per decryption (Step 2) while r_0, r_1, r_2 and r_3 are subject to a single select operation per decryption (Step 1). The output cardinalities of Steps 4 and 5 are shown within operation 8 and 9 in Figure 10. There is a dramatic increase in the output cardinality of Step 4, since this step involves the Cartesian product operation. As expected, the last step shows a drastic fall in cardinality, since each tuple of the input relation is subject to seven tests for equality/equivalence. With Algorithm 5, 90% of the keys were uniquely retrieved (remaining have 10 possible keys) with 6 decryptions. With 7 decryptions, the AES keys for all samples were uniquely deduced. This compares quite favourably with Algorithm 4 which requires about 17 decryptions.

6. Discussion

In this section, we discuss further optimizations together with possible extensions. We also highlight the limitations of our approach and conclude with countermeasures against our attack.

6.1. Further Optimizations and Performance Bounds

In practice, it is possible to optimize the First Round Attack. In the current implementation, we consider the accesses in $\rho_{0-1,*,*}$ (the union of the elements in the first two runs) whenever the total number of accesses in the first run was less than 16 (since each round of AES makes 16 table accesses). Using accesses in the first run separately could give more precise and early information on some of the high-order nibbles and reduce the "noise" in some of the histograms.

The number of encryptions/decryptions required for Algorithms 2, 3, 4 and 5 are respectively $\frac{-16}{\log_2 c}$, $\frac{-4}{\log_2 c}$, $\frac{-8}{\log_2 c}$ and $\frac{-4}{\log_2 c}$. Is there a lower bound on the number of encryptions/decryptions required to recover the key? To answer this, imagine a hypothetical relation with the low-order nibbles of the 16 bytes of the AES key as attributes. Upon initialization, the cardinality of this relation is 2^{64}. Now if the 16 equations were used as selection predicates in the decryption of δ blocks of ciphertext, then the size of the output relation would be $2^{64} \times c^{16\delta}$. Equating to 1, we get $\delta = \frac{-4}{\log_2 c}$. Algorithm 3 (Scenario I) and Algorithm 5 (Scenario II) achieve this lower bound on number of encryptions and decryptions respectively.

6.2. Limitations

Our algorithms assume accurate reporting of cache accesses by the spies. False positives (i.e. spurious cache accesses) will have adverse fallout to the extent that the effective run size may be increased thus requiring additional encryptions/decryptions. On the other hand, even a single false negative in the accesses reported by the spy threads and used by our algorithm will result in elimination of the correct key.

As with many access-driven attacks, we do assume that the victim and multi-threaded spy process are on the same processor core as a determined attacker may be able to co-locate itself with the victim. For example, in a multi-user environment, a user could simply request inordinate CPU resources and obtain access to multiple cores including the one victim is running on. We also assume that the core running the victim and spy do not simultaneously host another active process running AES. Otherwise, the accesses made by the latter may be mistaken for accesses by the victim possibly leading to flawed conclusions.

In Scenario I, plaintext blocks supplied to our key retrieval algorithms should not be related. Two or more plaintexts differing in only a few bits or bytes provide little new information to our algorithms. So, in this case, it will be necessary to use a larger number of encryptions to deduce the AES key.

6.3. Enhancements and Extensions

In this work, we have targeted OpenSSL version 0.9.8a which supports five AES tables. Some versions of OpenSSL use only four tables (for example, versions 1.0.0p and 1.0.2a). Four tables are sufficient for encryption (but not decryption). In this paper, access to the fifth table provides synchronization. With only four tables, it is necessary to design heuristics that identify the start of a new encryption.

All the processors used by us support hardware pre-fetching (i.e. on a cache miss, the next line is pre-fetched in anticipation of its future access). Pre-fetching is the default option and was disabled in our experiments. With pre-fetching enabled, the spies will be unable to distinguish between cache lines fetched by the victim and those pre-fetched by the processor. Worse still, a spy thread will inadvertently cause the processor to pre-fetch the line next to the one it just accessed. Consequently, the spy thread may end up finding that all cache lines were "accessed" by the victim.

It is necessary to design and implement algorithms that work in an environment where hardware pre-fetching is enabled. Likewise, it is necessary to modify our algorithms so that they are error-tolerant. Initial experiments with error tolerant algorithms and pre-fetching lead us to believe that our attack can be adapted to handle errors and hardware pre-fetching albeit at the cost of a larger number of encryptions/decryptions.

Most of the Intel caches have block size = 64 bytes. However, the IBM Power PC processor has block size = 128 bytes. Our attack can be adapted to work on the latter. The first round attack will obtain the first three bits of each byte of the AES key. To obtain the remaining five bits (second round), we will need much larger tables and so our key-retrieval algorithms will take more time. More importantly, the acceptable run size will be half that in the case with block size = 64 bytes (the compression ratio will be $\frac{|\rho_{*,*,*}|}{8}$ instead of $\frac{|\rho_{*,*,*}|}{16}$). This will necessitate more CPU interruptions.

Another possible direction is to operationalize our attack so that the espionage software provides to our algorithms, both, the sets of cache line numbers accessed by the victim as well as the blocks of ciphertext for further processing. Whether a "trojanized" version of these pieces can be created is material for further investigation.

6.4. Countermeasures

The multiple tables used in the software implementation of AES have much redundancy. For example, the i^{th} entries in tables T_0–T_3, $0 \leq i \leq 255$ are the same save for a permutation of their bytes. Also, the entries in T_4 are all contained in T_0. The single table is the default implementation in some versions of OpenSSL. This could thwart side channel attacks for run sizes experimented with in this paper. However, for very small run sizes (4-5 accesses per run), our attack may still be successful. [6] uses a single 256-byte table which occupies only four cache lines. The code that implements AES encryption/decryption may not be as efficient as in the case with five tables. On the other hand, the attack described here may either be unsuccessful or may require many more encryptions. Other mitigation strategies include the pre-loading of AES tables before each round and the addition of spurious accesses. These could lead to errors in the accesses reported by the spies but could be handled by error-tolerant key retrieval algorithms. Many more countermeasures have been proposed in [4], [11] and [12].

The most effective countermeasure is to support AES in hardware. Most Intel x86 processors beginning with the Westmere family include the AES-NI [13] instructions. Since the hardware implementation does not use processor cache to store the lookup tables, the attack described here will not work. However, some processors like Core 2 Duo with an installed base that is not insignificant do not have hardware support for AES as also the Pentium and Celeron models within the Westmere family.

7. Related Work

It was first mentioned by Hu [14] that cache memory can be considered as a potential vulnerability in the context of covert channels to extract sensitive information. Later Kocher [15] demonstrated the data-dependent timing response of cryptographic algorithms against various public-key systems. Based on his work, [16] mentioned the prospects of using cache memory to perform attacks based on cache hits in S-box ciphers like Blowfish. One formal study of such attacks using cache misses was conducted in [17].

Access-driven cache attacks were reported in [18] on RSA for multithreaded processors. Osvik et al. proposed an approach [19] and analysis for the access-driven cache attacks on AES for the first two rounds. They introduced the Prime+Probe [6] technique for cache attacks. In the Prime phase, the attacker fills the cache with its own data before the encryption. In the Probe phase, it accesses its data and determines whether each access results in a hit or miss. In the synchronous version of their attack, 300 encryptions were required to infer a 128-bit AES key on Athlon64 platform whereas in the asynchronous attack, which required statistical data on the frequency of accessed cache lines, they retrieved 45.7 bits of the 128-bit AES key after one minute.

The ability to detect whether a cache line has been evicted or not was further exploited by Neve et al. [20]. They designed an improved access-driven cache attack on the last round of AES on a single threaded processor. Aciiçmez et al. [21] presented a realistic access-driven cache attack by targeting I-cache attack based on vector quantization and hidden Markov models on OpenSSL's DSA implementation.

Gullasch et al. [7] proposed an efficient access-driven cache attack when attacker and victim use a shared crypto library. The spy process first flushes the AES lookups tables from all levels of the cache and interrupts the victim process after allowing it a single lookup table access. After every interrupt, it calculates the reload time to find which memory line is accessed by the victim. This information is further processed using a neural network to remove noise. Their experimental measurement method is similar to ours but the key retrieval is very different. Gullasch et al. attack is a practical and real time attack on AES-128. However, it requires frequent interruptions to the victim and requires about 100 encryptions. Its advantages are that it does not require synchronization nor does it requires the knowledge of the plaintext or ciphertexts.

Extending the work of Gullasch et al., [22] conducted a cross-core attack on the Last Level Cache (L3 on processors with three levels of cache) executing the spy and the victim concurrently on two different cores. They introduced Flush+Reload technique which is effective across multiple processor cores and virtual machine boundaries.

Trace-driven cache attacks were first theoretically introduced in [17]. They proposed a chosen plaintext attack on DES which required 2^{10} blocks of plaintext to collect cache profiles and 2^{32} computational steps to recover the key. Gallais et al. [23] proposed an improved adaptive known plaintext attack on AES implemented for embedded devices. Their attacks recover a 128-bit AES key with exhaustive search of at most 230 key hypotheses.

Aciiçmez et al. [5] provided an analytical treatment of trace-driven cache attacks and analyzed its efficiency against symmetric ciphers. Trace-driven cache attacks were further investigated by Zhao et al. [24] on AES and CLEFIA by considering cache misses and S-box misalignment.

Tsunoo et al. [25] pioneered the work on time-driven cache attacks by observing that the cache access pattern caused timing variations. Their attack comprises of two processes: obtaining the key differences and collecting cache timing data. They demonstrated that DES could be broken using 2^{23} known plaintexts and 2^{24} calculations at success rate $> 90\%$ on 600 MHz Pentium III processor. A similar approach was used by Bonneau and Mironov [12] where they emphasized individual cache collisions during encryption instead of overall hit ratio. Although this attack was a considerable improvement over previous work, it still requires 2^{13} timing samples.

Bernstein provided a practical cache attack [4] that can be categorized as a remote time-driven cache attack. He presented a known plaintext attack on a remote server running AES encryption. [26] is a recent work applying Bernstein's attack on ARM Cortex-A platform used on Android-based systems.

Neve [27] revisited Bernstein's attack technique and explains why his attack works. Concurrently but independently to the work of Bernstein [4], Osvik et al. [6] also described a similar time-driven attack with Evict+Time technique in which an attacker evicts cache lines from all levels of the cache and then identifies those which are accessed during the encryption.

A similar attack was proposed by Aciiçmez et al. [28] that was extended to use second round information of the AES encryption. Following the work of Tsunoo et al. [25], Canteaut et al. [29] proposed another variant of Bernstein's attack in which they described the influence of the cache initial state and cache parameters. Tiri et al. proposed an analytical model for forecasting the strength of symmetric ciphers by last round correlation attack [30] on AES. Further time-driven cache attacks were investigated by [31], [32].

Zhang et al. [33] targeting virtualized environments extract the private ElGamal key of a GnuPG description running in the scheduler of the Xen hypervisor [34]. Weiß et al. [35] used Bernstein's timing attack on AES running inside an ARM Cortex-A8 single core system in a virtualized environment to extract the AES encryption key. Irazoqui et al. [36] performed Bernstein's cache based timing attack in a virtualized environment to recover the AES secret key from co-resident VM with 2^{29} encryptions. Later Irazoqui et al. [37] used a Flush + Reload technique and recovered the AES secret key with 2^{19} encryptions.

8. Conclusions

We designed and implemented algorithms to recover the 128-bit AES key in two scenarios – when either the plaintext is known or when the ciphertext is known. It was assumed that, both, spy and victim (process computing AES encryptions/decryptions) are co-located on the same processor core and that their executions are interleaved. In both cases, a multi-threaded spy process collects a set of cache line numbers of AES table entries accessed by the victim and presents it to our algorithms for further processing. Our algorithms are concise and elegantly expressed using relational algebraic operations. An unoptimized implementation of our algorithm runs in under a minute.

With sets of cache line numbers provided by the spy threads, our experimental setup was able to recover the full 128-bit AES key using only 5-7 encryptions/decryptions in both Scenario I and Scenario II. Finally, we also presented analytical models that accurately predicted the number of encryptions/decryptions required.

We believe that this is probably the first successful attempt at retrieving the AES key with so few encryptions/decryptions in a practical setting.

References

[1] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[2] Y. Zhou and D. Feng, "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing." 2005. [Online]. Available: http://eprint.iacr.org/2005/388

[3] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer Science & Business Media, 2002.

[4] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[5] O. Acıiçmez and Ç. K. Koç, "Trace-driven Cache Attacks on AES," in *Proceedings of the 8th International Conference on Information and Communications Security*, ser. ICICS'06. Berlin: Springer, 2006, pp. 112–121. [Online]. Available: http://dx.doi.org/10.1007/11935308_9

[6] D. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology CT-RSA 2006*, ser. Lecture Notes in Computer Science, D. Pointcheval, Ed. Springer, 2006, vol. 3860, pp. 1–20.

[7] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.22

[8] B. Roy, R. P. Giri, Ashokkumar C., and B. Menezes, "Design and Implementation of an Espionage Network for Cache-based Side Channel Attacks on AES," in *Proceedings of the 12th International Conference on Security and Cryptography*, 2015, pp. 441–447. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005576804410447

[9] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill, 2011. [Online]. Available: https://books.google.co.in/books?id=Hvn_QQAACAAJ

[10] A. Swami and K. B. Schiefer, "On the Estimation of Join Result Sizes," in *Proceedings of the 4th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '94. New York, USA: Springer, 1994, pp. 287–300. [Online]. Available: http://dl.acm.org/citation.cfm?id=188573.188621

[11] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, p. 52, 2006.

[12] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks Against AES," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Springer, 2006, vol. 4249, pp. 201–215.

[13] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough AES Performance with Intel® AES New Instructions," *White paper, June*, 2010.

[14] W.-M. Hu, "Lattice scheduling and covert channels," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP '92. Washington, DC, USA: IEEE Computer Society, 1992, pp. 52–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=882488.884165

[15] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '96. London, UK: Springer, 1996, pp. 104–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=646761.706156

[16] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *J. Comput. Secur.*, vol. 8, pp. 141–158, Aug. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=1297828.1297833

[17] D. Page, "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel." *IACR Cryptology ePrint Archive*, vol. 2002, p. 169, 2002.

[18] C. Percival, “Cache missing for fun and profit,” 2005.

[19] E. Tromer, D. Osvik, and A. Shamir, “Efficient Cache Attacks on AES and Countermeasures,” pp. 37–71. [Online]. Available: http://dx.doi.org/10.1007/s00145-009-9049-y

[20] M. Neve and J.-P. Seifert, “Advances on Access-Driven Cache Attacks on AES.” in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, E. Biham and A. Youssef, Eds. Springer, 2007, vol. 4356, pp. 147–162.

[21] O. Acıiçmez, B. B. Brumley, and P. Grabher, “New results on instruction cache attacks,” in *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer, 2010, pp. 110–124.

[22] Y. Yarom and K. E. Falkner, “Flush+ Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.” *IACR Cryptology ePrint Archive*, vol. 2013, p. 448, 2013.

[23] J. F. Gallais, I. Kizhvatov, and M. Tunstall, “Improved trace-driven cache-collision attacks against embedded AES implementations,” in *Information Security Applications*. Springer, 2011, pp. 243–257.

[24] X. J. Zhao and T. Wang, “Improved Cache Trace Attack on AES and CLEFIA by Considering Cache Miss and S-box Misalignment.” *IACR Cryptology ePrint Archive*, vol. 2010, p. 56, 2010.

[25] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, “Cryptanalysis of DES Implemented on Computers with Cache.” in *Cryptographic Hardware and Embedded Systems - CHES 2003*, ser. Lecture Notes in Computer Science. Springer, 2003, vol. 2779, pp. 62–76.

[26] R. Spreitzer and B. Gérard, “Towards more practical time-driven cache attacks,” in *Information Security Theory and Practice. Securing the Internet of Things*. Springer, 2014, pp. 24–39.

[27] M. Neve, J.-P. Seifert, and Z. Wang, “A refined look at Bernstein’s AES side-channel analysis,” in *Proceedings of the ACM Symposium on Information, computer and communications security*. ACM, 2006, pp. 369–369. [Online]. Available: http://doi.acm.org/10.1145/1128817.1128887

[28] O. Acıiçmez, W. Schindler, and Ç. K. Koç, “Cache based remote timing attack on the AES,” in *Topics in Cryptology – CT-RSA 2007*. Springer, 2006, pp. 271–286. [Online]. Available: http://dx.doi.org/10.1007/11967668_18

[29] A. Canteaut, C. Lauradoux, and A. Seznec, “Understanding cache attacks,” Research Report RR-5881, 2006. [Online]. Available: https://hal.inria.fr/inria-00071387

[30] K. Tiri, O. Acıiçmez, M. Neve, and F. Andersen, “An analytical model for time-driven cache attacks,” in *Fast Software Encryption*. Springer, 2007, pp. 399–413.

[31] C. Rebeiro, M. Mondal, and D. Mukhopadhyay, “Pinpointing cache timing attacks on AES,” in *23rd International Conference on VLSI Design*. IEEE, Jan 2010, pp. 306–311.

[32] A. C. Atici, C. Yilmaz, and E. Savas, “An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks,” in *IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C)*. IEEE, 2013, pp. 74–83.

[33] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM side channels and their use to extract private keys,” in *Proceedings of the ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[34] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou, “Hardware-software integrated approaches to defend against software cache-based side channel attacks,” in *IEEE 15th International Symposium on High Performance Computer Architecture.* IEEE, 2009, pp. 393–404.

[35] M. Weiß, B. Heinz, and F. Stumpf, “A cache timing attack on AES in virtualization environments,” in *Financial Cryptography and Data Security*. Springer, 2012, pp. 314–328.

[36] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, “Fine grain Cross-VM Attacks on Xen and VMware are possible!” *IACR Cryptology ePrint Archive*, vol. 2014, p. 248, 2014.

[37] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, “Wait a minute! A fast, Cross-VM attack on AES,” in *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 299–319.