Sādhanā (2019) 44:88
https://doi.org/10.1007/s12046-019-1070-8

An error-tolerant approach for efficient AES key retrieval in the presence of cacheprefetching – experiments, results, analysis

C ASHOKKUMAR[1,*], M BHARGAV SRI VENKATESH[2], RAVI PRAKASH GIRI[1], BHOLANATH ROY[1] and BERNARD MENEZES[1]

[1]Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India
[2]Department of Electrical Engineering, Indian Institute of Technology Bombay, Mumbai, India
e-mail: ashokkumar@cse.iitb.ac.in; sri@ee.iitb.ac.in; raviprakash@cse.iitb.ac.in; bholanath@cse.iitb.ac.in; bernard@cse.iitb.ac.in

MS received 23 April 2018; revised 18 October 2018; accepted 20 November 2018; published online 21 March 2019

Abstract. The challenge in cache-based attacks on cryptographic algorithms is not merely to capture the cache footprints during their execution but to process the obtained information to deduce the secret key. Our principal contribution is to develop a theoretical framework based upon which our AES key retrieval algorithms are not only more efficient in terms of execution time but also require up to 75% fewer blocks of ciphertext compared with previous work. Aggressive hardware prefetching greatly complicates access-driven attacks since they are unable to distinguish between a cache line fetched on demand versus one prefetched and not subsequently used during a run of a victim executing AES. We implement a multi-threaded spy code that reports accesses to the AES tables at the granularity level of a cache block. Since prefetching greatly increases side-channel noise, we develop sophisticated heuristics to "clean up" the input received from the spy threads. Our key retrieval algorithms process the sanitized input to recover the AES key using only about 25 blocks of ciphertext in the presence of prefetching and, stunningly, a mere 2–3 blocks with prefetching disabled. We also derive analytical models that capture the effect of varying false positive and false negative rates on the number of blocks of ciphertext required for key retrieval.

Keyword. AES; access-driven; cache attacks; side channel; table look-up.

1. Introduction

Side-channel attacks leak sensitive cryptographic information through physical channels such as power, timing, etc. and typically are specific to the actual implementation of the cryptographic algorithm [1]. An important class of these attacks is based on obtaining access time measurements from cache memory systems. The Advanced Encryption Standard (AES) [2] is a widely adopted algorithm for secret key cryptography. Most software implementations of AES, including those in the cryptographic library OpenSSL [3], make extensive use of cache-resident table look-ups in lieu of time-consuming mathematical field operations.

Many 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 [4]. The attacker (or spy) and the victim use a single copy of the same cryptographic library and its binaries are mapped to the virtual spaces of attacker and victim. In a Flush and Reload attack [5], for example, the spy process flushes out the AES tables from cache. When the victim is scheduled, it brings in some of the evicted lines. When the control returns back to the spy, it determines which of the evicted lines were fetched 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. Inputs obtained by the spy are then analysed to compute the AES key.

Our algorithms to deduce the AES key use (i) several blocks of ciphertext encrypted with the same key and (ii) the set of line (or block) numbers of AES table entries accessed by the victim during the decryption of each of those blocks of ciphertext. Our use of ciphertext (rather than plaintext as in some previous works) is far more realistic since the former is readily available. Several entries in the AES table are placed on a single cache line and the spy provides a set (not list) of lines accessed. The absence of spatial information (the specific table entry on a line required by the victim) and temporal information (the order of accesses) makes it challenging to deduce the key, especially for sets with larger cardinalities.

*For correspondence

The goal of this work is the retrieval of the AES key with the fewest possible blocks of ciphertext. The attack in [6] was implemented with a similar goal. However, the presence of false negatives in the spy input defeats their attack. Moreover, cache prefetching (described later) was disabled in those experiments. As in [6, 7], we derive 16 equations that relate the output of a round to blocks of ciphertext and bits of the AES round key. Our principal contribution is the development of a theoretical framework upon which our AES key retrieval algorithms are based. The latter are not only more efficient in terms of execution time but also require up to 75% fewer blocks of ciphertext compared with previous work. The dramatic increase in efficiency of our approach and its error tolerance is attributed, in part, to our exploitation of the peculiar structure of these equations and extraction of the maximum possible information from them. The key retrieval algorithms are agnostic with respect to cache attack strategy ("Flush and Reload" [5], "Prime and Probe" [8], etc.), to specific software implementations (for example, single/multiple look-up tables [3]), to single core versus multi core attack scenarios and to noise volume on the cache side channel.

The complexity of cache attacks is exacerbated by hardware prefetchers [9] (implemented internally by processors to reduce memory latency). When a cache miss occurs, the processor not only fetches the missed line but also the next or previous line in anticipation of its use. Processors targeted in earlier attacks [8, 10] employed stride prefetchers while modern processors sport far more sophisticated prefetchers. The latter use history tables to record patterns of cache miss addresses, make predictions of future misses and perform aggressive prefetching based on their predictions [11]. The substantial increase in false positive rate (due to lines prefetched but not subsequently used during a run of the victim executing AES) has a serious adverse effect on the efficiency of access-driven attacks.

Our efforts with prefetching turned off enabled us to retrieve the AES key with a mere 2–3 blocks of ciphertext – two orders of magnitude better than the best result obtained so far. The natural question to be asked then is with prefetching turned on, what is the number of decryptions required? Our main contribution is to answer this question by implementing (i) prefetching-aware multithreaded spy code, (ii) heuristics to sanitize the input received from the spy threads and (iii) key retrieval algorithms that operate on the sanitized input. The spy code was ported on to the Intel Core 2 Duo, Core i3, Core i5 and Core i7 (the latter three with very aggressive prefetchers [9]). Our heuristics and key retrieval algorithms succeeded in retrieving the entire AES key with only about 25 blocks of ciphertext.

Our second contribution is a systematic investigation into the effect of errors on the number of decryptions (ciphertext blocks) required. False positives and false negatives in the input provided by the spy threads are caused, in part, by measurement errors. Moreover, because of the way the spy threads are designed to operate, there is a substantial rise in false negatives besides the expected increase in false positives. All of our experiments were conducted in a lab setting with few other processes sharing the core in addition to the victim and spy. In more realistic scenarios, we would expect even greater noise and hence higher error rates. We develop analytical models to study the effect of varying false positive and false negative rates on the number of decryptions required and compare these to actual results from our experiment set-up. This work is the first in attempting a quantitative investigation of the effect of noise on the success probability of key retrieval.

This paper is organized as follows. Section 2 summarizes work related to the theme of this paper. Section 3 contains an introduction to AES and, in particular, its software implementation. Our analytical models were corroborated through experiments – the experimental set-up, spy code and key retrieval strategy are also described in section 3. Sections 4 and 5 present the algorithms, model and results related to the First Round and Second Round Attacks, respectively. Section 6 discusses other issues of relevance, including limitations and countermeasures, and section 7 contains the conclusion.

2. Related work

It was first mentioned by Hu [12] that cache memory can be considered as a potential vulnerability in the context of covert channels to extract sensitive information. Later Kocher [13] demonstrated the data-dependent timing response of cryptographic algorithms against various public-key systems. Based on his work, [14] 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 [15].

Access-driven cache attacks were reported in [16] on RSA for multithreaded processors. Tromer *et al* [8] proposed an approach and analysis for the access-driven cache attacks on AES for the first two rounds. They introduced the Prime+Probe 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 the Athlon64 platform.

The ability to detect whether a cache line has been evicted or not was further exploited by Neve *et al* [17]. They designed an improved access-driven cache attack on the last round of AES on a single-threaded processor. Aciiçmez *et al* [18] presented a realistic access-driven cache attack targeting I-cache based on vector quantization

and hidden Markov models on OpenSSL's DSA implementation.

Gullasch *et al* [10] proposed an efficient access-driven cache attack when attacker and victim use a shared crypto library. Their experimental measurement method is similar to ours but their key retrieval is very different. It does not require synchronization or knowledge of the plaintexts or ciphertexts. They retrieved the AES key using about 100 blocks of ciphertext. Extending the work of Gullasch *et al* [10], Yarom and Falkner [5] conducted a cross-core attack on the LLC (Last Level Cache, L3 on processors with three levels of cache) with the spy and the victim executing concurrently on two different cores. They introduced the Flush+Reload technique, which is effective across multiple processor cores and virtual machine boundaries. The first access-driven cache attack on smartphones was proposed in [19] on misaligned AES T-tables.

Acıiçmez and Koc [20] provided an analytical treatment of trace-driven cache attacks and analysed its efficiency against symmetric ciphers. Trace-driven cache attacks were first theoretically introduced in [15]. Gallais *et al* [21] 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. Trace-driven cache attacks were further investigated by Zhao and Wang [22] on AES and CLEFIA by considering cache misses and S-box misalignment.

Tsunoo *et al* [23] pioneered the work on time-driven cache attacks. They demonstrated that DES could be broken using 2^{23} known plaintexts and 2^{24} calculations. A similar approach was used by Bonneau and Mironov [24], 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 required 2^{13} timing samples. Bernstein [25] presented a practical 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 *et al* [27] revisit Bernstein's attack technique and explain why his attack works. Concurrent to but independent of the work of Bernstein [25], Osvik *et al* [28] introduced the Evict+Time technique in which an attacker evicts cache lines from all levels of the cache and then identifies those that are accessed during the encryption. Other time-driven attacks were investigated by [23, 29–33].

Irazoqui *et al* [34] introduced a new shared LLC attack that works across cores and VM boundaries. Their attack is implemented on the basis of Prime+Probe technique enabled by huge pages without de-duplication. A similar LLC attack was proposed in [35] on various versions of GnuPG. Another attack on LLC was implemented in [36] that does not require the use of large pages.

Zhang *et al* [37], targeting virtualized environments, extract the private ElGamal key of a GnuPG description running in the scheduler of the Xen hypervisor [38]. Weiß *et al* [39] 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* [40] 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* [41] used a Flush+Reload technique and recovered the AES secret key with 2^{19} encryptions.

Hardware prefetchers, first discussed in [42], aim to provide temporal locality by bringing data into the cache, complicating the various cache-based attacks as also observed in [10, 28, 43]. Tromer *et al* [8] discussed a pointer chasing technique in which the attacker's memory is organized into a linked list. They mounted Prime+Probe-based attack by traversing the list to avoid the read-ahead of memory by the hardware prefetcher.

Liu *et al* [44] discusses an LLC attack on ElGamal decryption in which the authors randomly organize the memory lines in each eviction set as a linked list. Their random permutation prevented the hardware prefetching memory lines in the eviction set. Recently, [45] described a novel prefetch side-channel attack exploiting weakness of prefetch instructions to obtain address information by defeating SMAP, SMEP and kernel ASLR. The attack is based on the primitive that prefetch can be used to fetch inaccessible privileged memory into caches without performing any privilege checks.

Fuchs and Lee [46] presented a randomized set-balanced policy able to disrupt the cache footprint construction and thwart Prime+Probe cache attacks. They suggested two extensions to the conventional prefetching scheme – randomized prefetching policy to operate at different levels of aggressiveness and a set balancer to balance the load across all cache lines. Hardware prefetching has been discussed further in [16, 47].

3. Background and attack preliminaries

We first summarize the key points in the software implementation of AES and then describe the espionage infrastructure designed by us. Finally, the strategy used for key retrieval is explained. The notations used are described in table 1.

3.1 *AES preliminaries*

AES is a symmetric key block cipher whose construction is based on 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 ciphertext to 128

Table 1. Notations.

Notation	Explanation
k_i	i^{th} byte of 10^{th} round key of AES
c_i, $c_{i,j}$	i^{th} byte of ciphertext (in j^{th} block)
T_t	AES T-table, $0 \le t \le 4$
y'	high-order nibble of byte y
y''	low-order nibble of byte y
$x_{i,t,r,d}$	table index of i^{th} access to T_t in Round r of Decryption d
x_j	equivalent to $x_{i,t,r,d}$ where $j = t + 4i, r = 2$ and d is dropped for brevity
$G_{i,t,r,d}$	set of guesstimates of the line number corresponding to i^{th} access to T_t in Round r of Decryption d
$P(A)$	probability of event A
f_q	average false negative rate in set of guesstimates after q refinements
$\mid G \mid_q$	average cardinality of the set of guesstimates after q refinements
δ	number of decryptions considered

bits of plaintext, during decryption. The 16-byte ciphertext $C = (c_0, c_1, \ldots, c_{15})$ may be thought of as being arranged column-wise in a 4×4 array of bytes. Each byte (represented as two hex characters) is an element in the binary field, $GF(2^8)$. This "state array" gets transformed after each step in a round. At the end of the last round, the state array contains the plaintext.

In decryption, all rounds except the last involve four steps – InvSubBytes (inverse byte substitution), InvShiftRows (inverse row shift), InvMixColumns (inverse column mixing) and a round key addition (the last round skips the inverse column mixing step). The substitution and column mixing steps are defined using algebraic operations over the field $GF(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. For example, in the inverse column mixing step, the state array is pre-multiplied by the following matrix B^{-1}, where B^{-1} is the inverse of the matrix B used in the column mixing step of encryption:

$$B^{-1} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}, \quad B = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}.$$

Like a block of plaintext, ciphertext and the state array, the 128-bit AES key, the original 16-byte secret key K is arranged columnwise in a 4×4 array of bytes. It is used to derive 10 different round keys to be used in the round key operation of each round. The round keys are denoted as $K^{(r)}, r = 1, 2, \ldots, 10$. In a software implementation, field operations are replaced by relatively inexpensive table look-ups, thereby speeding encryption and decryption.

In the versions of OpenSSL targeted in this paper, five tables are employed (each of size 1 KB). A table T_t, $0 \le t \le 4$, 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. The output of Round r $(r = 1, \ldots, 9)$ is obtained from the input using 16 table look-ups (4 per table T_t, $0 \le t \le 3$) and 16 XOR operations as shown in (1)–(4):

$$\left(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}\right) \leftarrow T_0\left[x_0^{(r)}\right] \oplus T_1\left[x_{13}^{(r)}\right] \oplus T_2\left[x_{10}^{(r)}\right] \oplus T_3\left[x_7^{(r)}\right] \oplus K_0^{(r)}, \tag{1}$$

$$\left(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}\right) \leftarrow T_0\left[x_4^{(r)}\right] \oplus T_1\left[x_1^{(r)}\right] \oplus T_2\left[x_{14}^{(r)}\right] \oplus T_3\left[x_{11}^{(r)}\right] \oplus K_1^{(r)}, \tag{2}$$

$$\left(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}\right) \leftarrow T_0\left[x_8^{(r)}\right] \oplus T_1\left[x_5^{(r)}\right] \oplus T_2\left[x_2^{(r)}\right] \oplus T_3\left[x_{15}^{(r)}\right] \oplus K_2^{(r)}, \tag{3}$$

$$\left(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}\right) \leftarrow T_0\left[x_{12}^{(r)}\right] \oplus T_1\left[x_9^{(r)}\right] \oplus T_2\left[x_6^{(r)}\right] \oplus T_3\left[x_3^{(r)}\right] \oplus K_3^{(r)}. \tag{4}$$

Here $K_j^{(r+1)}$ refers to the j^{th} column of round key $K^{(r+1)}$.

In the last round, table T_4 is used instead of $T_0, \ldots, T_3$ due to the absence of the column mixing step. The value returned by the table look-up is XORed with the corresponding byte of the round key. Since each round involves 16 table accesses, a complete encryption/decryption using 128-byte key involves a total of 160 table accesses.

For performance reasons, the tables reside in cache. The granularity of transfer between the cache and main memory is a line that is 64 bytes on all machines we experimented with. Each of $T_0 - T_3$ contains 256 4-byte elements. Hence, 16 elements reside on a single cache line and each of $T_0 - T_3$ occupies 16 lines. An index into a table is 8 bits – the high-order nibble specifies the line within the table and the low-order nibble specifies the element within a line.

3.2 *Experimental set-up*

Executions of the spy process and the victim performing decryptions are interleaved on the same core. Ideally, the victim (V) makes a few AES table accesses during its run before being preempted. The spy, scheduled next, then

attempts to infer the line numbers of the table elements accessed in the preceding run of V. In most operating systems (OSs), however, the time slice provided to an executing process is about a millisecond. This is sufficient for V to decrypt tens of blocks of ciphertext, making it impossible for the spy to deduce the decryption key. What is needed is a way to restrict the duration of V's time slice to about 1100 ns.

Similar to [10], we employ a multi-threaded spy process. The spy program creates a high-resolution POSIX timer (used by all the spy threads) and an array of binary semaphores – $S[i]$ is the semaphore associated with Thread i. All but one of the semaphores are initialized to 0. Hence, all threads are blocked on their respective semaphores except for the one that is initialized to 1. The unblocked thread accesses a pre-determined list of cache lines to determine which of these have just been accessed by V. The exact list depends on whether prefetching is enabled and, if so, the type of prefetcher. The spy thread then flushes all the lines containing the AES tables from cache.

Spy Program

```
   Input: cacheLines[], δ
   Output: Set of cache lines accessed by Victim in
           previous run
1  timer_create(clock_id, sigevent, timerid)
2  Signal Handler:
3  sem_post(S[i+1])
4  Spy Thread T_i:
5  while # of decryptions < δ do
6      sem_wait(S[i])
7      for each cLine in cacheLines do
8          if accessTime[cLine] < THRESHOLD then
9              isAccessed[cLine] ← true
10         end
11         clflush(cLine)
12     end
13     newTimerValue ← t
14     timer_settime(timerid, NULL, newTimerValue,
         oldTimerValue)
15 end
```

The last task of the spy thread is to arm the timer by initializing its value to $t \sim 1100$ ns. It then returns to the first statement of the loop, wherein it blocks on its semaphore (Line 6). Now all the spy threads are blocked on their respective semaphores. Hence, V is scheduled next and it resumes performing decryptions. On expiration of the timer value, the kernel sends a signal to the signal handler, which unblocks Thread $i + 1$ (Line 3). V is preempted and Thread $i + 1$ is scheduled for the reason explained here.

Beginning with version 2.6.23 of the Linux kernel, the default process scheduler is the CFS (Completely Fair Scheduler). For each process or thread, the OS maintains a virtual run time that is a crude measure of CPU run time granted to that process. Scheduler fairness implies that V and the n spy threads each get roughly $1/(n+1)^{\text{th}}$ of the CPU time (ignoring other processes executing on the core). When a thread is blocked, its virtual run time remains fixed. The unblocked threads and processes get scheduled in turn and their virtual run times increase. Hence, when the blocked thread is unblocked its virtual run time is sufficiently low that V is preempted and the former gets rescheduled.

When a spy thread executes, it accesses lines of the AES tables in order. A cache miss results in the next (or previous) line being prefetched, causing the spy to wrongly infer that the prefetched line was accessed in the previous run of the victim. To defeat the effect of lines prefetched during the execution of the victim process, the number of accesses made by the victim during each of its runs should be minimized. This was accomplished by setting the time interval of the high-resolution POSIX timer to ~ 1100 ns on Intel Core i3/i5/i7 processors and ~ 1700 ns on Core 2 Duo.

One possible way to defeat prefetching during the execution of the spy is to access only the even-indexed lines (with a stride of 2). This strategy results in a high rate of false negatives since all odd-indexed lines are missed out. Worse still, the prefetcher is able to detect fixed strides in the access sequence and prefetch lines, thus defeating our strategy. We modify our approach by randomly generating all even numbers between 0 and 72 using the expression $(17^i \bmod 37) * 2$. This generates the sequence of line numbers 34, 60, 58, 24, 38, 54, ..., 2, 0, which are loaded into the array *cacheLines* in the spy program. With this modification, we obtained the correct set of even indices accessed by the victim on the Core 2 Duo since its prefetcher is now unable to detect a fixed stride.

This strategy fails on the Intel Core i3/i5/i7 since it incorporates a more aggressive prefetcher that tracks and remembers the forward and backward strides of the 16 most recently accessed 4-KB pages [9]. Hence, we modified the spy code to access 32 randomly selected pages between two consecutive accesses to the AES tables. Our spy program is versatile enough to handle the prefetchers in both Intel Core 2 Duo and Intel Core i3/i5/i7.

3.3 *Key retrieval strategy*

Key retrieval involves two steps. In the first, the high-order nibble of each byte of the AES key is obtained and in the second, the low-order nibbles are deduced. The two steps use inputs to the First and the Second Rounds; hence, they are referred to as First and Second Round Attacks, respectively.

Each round uses 16 equations. The LHS of an equation is the input to a round (except for the first round, this is also the output of the previous round). As mentioned in section 3.1, the LHS can also be thought of as an index to

an AES table. The RHS of an equation involves bytes of the ciphertext and the key. From the RHS of each equation, we identify a bunch of bits in the AES key (sub-key) whose true value we attempt to determine. The high-order nibble of the table index is the cache line number. The information provided by the spy threads is sets of line numbers. A preprocessing step is involved in converting this input to guesstimates of each cache line number accessed by the victim. Let $x_{i,t,r,d}$ be the table index and $G_{i,t,r,d}$ be the set of guesstimates of the line number corresponding to the i^{th} access to the t^{th} table in Round r of Decryption d.

Histograms are composed, one per equation; m sets of Bernoulli trials are conducted per histogram. Within a set, each sub-key value is assigned a score, 0 or 1 (success), which is determined as follows. The RHS of the equation is computed by substituting the sub-key value and ciphertext. If the high-order nibble of the computed byte matches a line number in the corresponding set $G_{i,t,r,d}$, the outcome is success. The sub-key value with the highest cumulative score is declared the winner and is assumed to be the actual sub-key.

Let X_c and $X_i, 1 \le i \le 2^s, i \ne c$, respectively, denote the random variables associated with the scores of correct and incorrect values of the sub-key and let p_c and p_{in} denote their success probabilities. Here 2^s is the number of possible values of the s-bit sub-key. These variables are binomially distributed, i.e., $X_c \sim B(p_c, m)$ and $X_i \sim B(p_{in}, m)$.

Lemma 1 *The probability $P_h(p_c, p_{in}, 2^s, m)$ that the score of the correct s-bit sub-key value is greater than those of all others after m Bernoulli trials is*

$$\sum_{n=0}^{m-1} \left\{ [P(X_c = n+1)][P(X_i \le n)]^{2^s-1} \right\}.$$

The next two sections describe the algorithms for the First and Second Round Attacks and include results from, both, experiments and a model.

4. First Round Attack

We first show how guesstimates are computed and then used in the algorithms for the First Round Attack.

4.1 *Obtaining the guesstimates*

Let $c_{0,j}, c_{1,j}, \ldots, c_{15,j}$ denote the 16 bytes of block j of the ciphertext and let $k_0, k_1, \ldots, k_{15}$ be the bytes of the 128-bit AES key. Before the start of the First Round, the ciphertext is XORed with the key, so the table indices are computed as

$$x_{i,t,1,d} = c_{t+4i,d} \oplus k_{t+4i}, \quad 0 \le i, t \le 3, \quad 1 \le d \le \delta. \qquad (5)$$

In the First Round Attack, we obtain the high-order nibbles of the AES key from the ciphertext and table line numbers as in

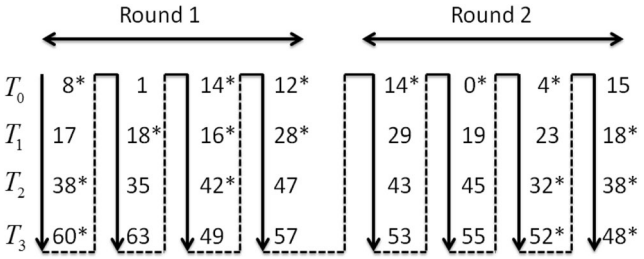

Figure 1. Sequence of table line numbers accessed.

$$k'_{t+4i} = x'_{i,t,1,d} \oplus c'_{t+4i,d}, \quad 0 \le i, t \le 3, 1 \le d \le \delta \qquad (6)$$

where x' and x'' denote, respectively, the high-order and low-order nibble of byte x. The first step in the attack is to obtain the 16 sets of guesstimates.

Figure 1 shows the exact sequence of line numbers accessed during the first two rounds of a decryption for a given key and ciphertext block. Table 2 shows the corresponding line numbers accessed by V in the first 10 runs as reported by the spy threads. For clarity of viewing, we represent line numbers in the range 0–63. Lines 0–15 are in T_0, 16–31 are in T_1, 32–47 are in T_2 and 48–63 are in T_3. Line 32 is the 4th even line number access in Round 2 but it appears early on in Run 3, potentially leading to the erroneous conclusion that it occurs in Round 1. Also, the same line number may appear in multiple consecutive runs even though it has been accessed just once.

A typical scenario encountered by V and the spy threads is as follows. During a run, V suffers one or more cache misses since all AES tables are flushed out by the previously executing spy thread. Modern processors employ out-of-order execution so that, in the event of a memory stall, later instructions are executed assuming no data dependences. Table access instructions further upstream cause more cache misses and requests to main memory. The first few missed lines are fetched into cache and processed by the CPU while a few others are placed in cache but not processed. Just then, V is preempted. When the next spy thread is scheduled, it reports the presence of several lines in cache. Of these, only the first few have actually been processed, so the remainder of those fetched will have to be re-fetched during the next run of V and so on, thus appearing superfluously in multiple runs of the spy input.

We attempt to eliminate redundancies in the spy input by deriving two auxiliary tables – an Addition Table and a Deletion Table (table 2). In the former, we include, for Run i, only the line numbers appearing in Run i but not Run $i-1$ of the spy input. In the Deletion Table, we include only line numbers missing in Run i but present in Run $i-1$ of the spy input.

The challenge is that the spies provide us with only the even line numbers and, then too, we are not sure of their exact positions. Moreover, a round is not aligned on a run boundary, so we cannot precisely identify the point at which the First Round accesses end and the Second Round

Table 2. Cache line numbers reported by consecutive spy threads.

Run	Spy Inputs				Deletion Table				Addition Table			
	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3
1	8		38	60					8		38	60
2	12,14	16,18	36,38,42	50,60	8				12,14	16,18	36,42	50
3	12,14	16,18,28	32,42,44	60			36,38	50		28	32,44	
4	12,14	16,28	42			18	32,44	60				
5	12,14	28	42			16						
6	0,12,14	28					42		0			
7	0,14	28			12							
8	0,4,14					28			4			
9	0,4		32	52	14						32	52
10	4	18	32,38	48,52	0					18	38	48

begins. One simple strategy is to include the fewest number of rows from the Addition Table at the start of a decryption so that the union of line numbers in those rows is 8 or more. The choice of 8 is because a round involves 16 table accesses; of these, 8 accesses, on average, would be to even line numbers.

Returning to table 2, we include the first two rows of the Addition Table to give us a total of 10 line numbers. From this, we compute the guesstimates as follows:

$$\begin{aligned} G_{i,0,1,d} &= \{8, 12, 14\} \\ G_{i,1,1,d} &= \{16, 18\} \\ G_{i,2,1,d} &= \{36, 38, 42\} \\ G_{i,3,1,d} &= \{50, 60\} \end{aligned} \qquad 0 \le i \le 3.$$

Note that line numbers 36 and 50 in $G_{i,2,1,d}$ and $G_{i,3,1,d}$, respectively, were incorrectly reported by the spy threads (they do not occur in the trace of Round 1 accesses in figure 1). We refer to such spurious accesses as false positives. Also, line number 28 was accessed but does not appear in $G_{i,1,1,d}$. Each missing element in a set of guesstimates is treated as a false negative with respect to the line number being guessed. Line 28 appears in the third row of the Addition Table. If we included the elements in this run to our guesstimates, then lines 32 and 44 (which are false positives) would also have to be added to $G_{i,2,1,d}$.

False positives and false negatives could be due to errors in the measurements made by the spy threads – for example, lines 36 and 50 are two such errors. On the other hand, they could be due to our preprocessing strategy being either too conservative or too liberal. With only two runs of the Addition Table, we excluded line 28. However, with three runs, we eliminated the false negative at the expense of two more false positives. Thus, there is, in general, a delicate tradeoff between reducing the number of false positives and reducing the number of false negatives.

4.2 *Key recovery algorithm*

Having computed the guesstimates, we next introduce an algorithm to recover the high-order nibbles of each byte of the AES key given δ blocks of ciphertext.

Algorithm 1 builds 16 histograms, one per high-order nibble of the AES key. We start with histograms 0, 4, 8 and 12 constructed from the Guesstimates $G_{i,0,1,d}, 0 \le i \le 3$, containing line numbers of T_0. After δ decryptions (a decryption is associated with a Bernoulli trial for each of the 64 nibble values, 16 per histogram), we identify the nibble value across all four histograms with the highest score. In the event of a tie, the value with the most convincing lead over the others in that histogram is selected. We declare it to be the true value of that nibble and the histogram bearing the winner is set aside. Let the winner be value y in $hist_{4m}$, so $k'_{4m} = y$. We then compute the corresponding line numbers accessed in T_0 using Eq. (6). Line number $y \oplus c'_{4m,1}$ is deleted from Guesstimates $G_{n,0,1,1}$ and $y \oplus c'_{4m,2}$ is deleted from $G_{n,0,1,2}$ and so on, where $0 \le n \le 3$ and $n \ne m$.

This procedure is repeated for the remaining three histograms (Line 4 of Algorithm 1), then for the remaining two and finally for the last histogram. At this point, key nibbles $k'_{4m}, 0 \le m \le 3$, are obtained. This is repeated for the histograms corresponding to each of the other tables (Line 1 of Algorithm 1) to retrieve all high-order nibbles of the key.

4.3 *Analysis and results*

We next analyse the performance of Algorithm 1 as a function of the false positive and false negative rates.

Consider key nibbles k'_{4m}, $0 \le m \le 3$. We refined the set of guesstimates after retrieval of every key nibble among the four considered. After every refinement, the average cardinality of the set of guesstimates decreases and the average false negative rates increase since some line numbers may be accessed more than once in a round.

Let f_q be the average false negative rate after q refinements to the set of guesstimates. It can be thought of as the probability that the line number accessed is not included in the set of guesstimates. The average cardinality of the set of guesstimates after q refinements, denoted as $\mid G \mid_q$, is a measure of the false positive rate per element[1] accessed. Let p_c and p_{in}, respectively, denote the probabilities of the correct and incorrect nibbles in a histogram receiving a boost. Hence

$$p_c = 1 - f_q, \quad 0 \leq q \leq 3 \tag{7}$$

and

$$p_{in} = f_q \left(\frac{\mid G \mid_q}{15} \right) + (1 - f_q) \left(\frac{\mid G \mid_q - 1}{15} \right). \tag{8}$$

Equation (8) follows from the fact the probability that an incorrect nibble receives a boost in case of a false negative is $\frac{|G|}{15}$, and is $\frac{|G|-1}{15}$ otherwise.

The probability that the correct value of a key nibble is the unrivalled top scorer in a histogram is computed from $P_h(p_c, p_{in}, 2^s, m)$ in Lemma 1 with $m = \delta$, the number of decryptions $2^s = 16$, the number of possible values of a key nibble and success probabilities p_c and p_{in} obtained, respectively, from Eqs. (7) and (8). Lines 12 and 13 of Algorithm 1 first attempt to find an unrivalled top scorer in at least one of four histograms, $hist_{4m}, 0 \leq m \leq 3$. The probability that this succeeds is $1 - (1 - P_h(p_c, p_{in}, 2^4, \delta))^4$.

The success probability of Algorithm 1 is given in the following theorem.

Theorem 1 *The probability of successfully recovering all 16 high-order nibbles of the AES key in δ decryptions is*

$$\left\{ \prod_{q=0}^{3} \left[1 - [1 - \mathcal{P}_h(f_q, \mid G \mid_q, 2^4, \delta)]^{4-q} \right] \right\}^4$$

where $\mathcal{P}_h(f_q, \mid G \mid_q, 2^4, \delta) = P_h(p_c, p_{in}, 2^4, \delta)$; f_q and $\mid G \mid_q$ are derived in Appendix I.

We performed experiments on an Intel Core i3 running Debian 8, Linux kernel 3.18 with OpenSSL version 1.0.2a. We generated 1000 samples – each sample comprises a randomly selected decryption key together with 50 random blocks of ciphertext. We ran each sample with 200 spy threads and created the set of guesstimates from the spy input. Using Algorithm 1, we attempted to deduce the AES key with varying number of ciphertext blocks.

Figure 2 shows the number of successes in 1000 samples. By success, we mean that all 16 high-order nibbles are recovered. In some cases, we recovered 15 nibbles successfully but a tie occurred in determining the last nibble. All of these cases were, however, easily resolved by closer inspection of the spy input. We included all such cases as successes. The unsuccessful case typically involved one or a few nibble values that were incorrectly guessed. With 16 decryptions, the success rate is 74%, but increases to 86% and 96% with 20 and 26 decryptions, respectively. For comparison, we included the unrefined version of Algorithm 1, which works on each histogram independently without updating the guesstimates. As shown in figure 2, the performance of the unrefined algorithm is considerably inferior to the refined one.

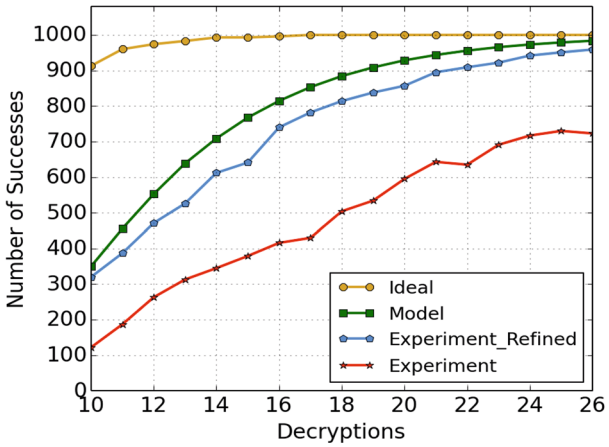

Figure 2. Number of successes per 1000 samples (Round 1).

Algorithm 1: First Round Attack

Input: Ciphertext, $c_{t+4i,d}$ and Guesstimates, $G_{i,t,1,d}$ $0 \leq i \leq 3$, $0 \leq t \leq 3$ and $1 \leq d \leq \delta$

Output: $k'_0, k'_1, ..., k'_{15}$ (high-order nibbles of each byte of the key)

1 **for** *each table,* T_t, $0 \leq t \leq 3$ **do**
2 $S_t = \{0, 1, 2, 3\}$
3 **while** S_t *is not empty* **do**
4 **for** *each* $i \in S_t$ **do**
5 // Compute histogram for key nibble, k'_{t+4i}
6 **for** *each decryption,* d, $1 \leq d \leq \delta$ **do**
7 **for** *each* $x \in G_{i,t,1,d}$ **do**
8 increment $hist_{t+4i}[x \oplus c'_{t+4i,d}]$
9 **end**
10 **end**
11 **end**
12 $j = argmax_{i \in S_t} hist_{t+4i}[y]$, $0 \leq y \leq 15$
13 $k'_{t+4j} = argmax_y hist_{t+4j}[y]$, $0 \leq y \leq 15$
14 $S_t = S_t - \{j\}$
15 **for** *each decryption,* d, $1 \leq d \leq \delta$ **do**
16 **for** *each* $i \in S_t$ **do**
17 $G_{i,t,1,d} = G_{i,t,1,d} - \{y \oplus c'_{t+4i,d}\}$
18 **end**
19 **end**
20 **end**
21 **end**

[1]The false positive rate per element is equal to $f_q \mid G \mid_q + (1 - f_q)(\mid G \mid_q - 1) = \mid G \mid_q + f_q - 1$.

We also ran the key retrieval algorithm on the same samples but assuming ideal guesstimates. $G_{i,t,1,d}$ is a set of ideal guesstimates if it contains line numbers only from T_t accessed in Round 1 of decryption d, and its only false negatives are the odd-numbered cache lines accessed in Round 1. This is an ideal experiment – one in which no measurement error is committed by the spy threads and in which the end of the First Round is unambiguously identified. $\mid G \mid_0$ and f_0 in the ideal case are, respectively, 1.82 and 0.5. With experimentally obtained spy inputs and the preprocessing strategy employed in deriving guesstimate sets, $\mid G \mid_0$ and f_0 increase to 2.15 and 0.55, respectively. As figure 2 shows, the success rate in the ideal case is nearly 100% with 14 decryptions. Thus the errors, whether induced by lapses in spy measurements or inherent in the preprocessing strategy, are seriously detrimental to performance.

5. Second Round Attack

The goal of the Second Round Attack is to obtain the low-order nibble of each byte of the key.

5.1 *Theoretical underpinnings*

From the algorithm used to generate round keys and by tracking how the input to Round 1 gets transformed to its output, we derived 16 equations shown here. They relate the second round inputs to the ciphertext and to various bytes of the key.

In the LHS of Eqs. (9)–(24), x_j represents an element in table T_t. For simplicity, we use a single subscript instead of $x_{i,t,r,d}$ used earlier. Here, the subscript j is equal to $t + 4i$. We use Round 2 accesses, so $r = 2$. Also, these equations are true for all the decryptions. In addition, we use c_{t+4i} in lieu of $c_{t+4i,j}$ (the second subscript j, the block number of ciphertext is suppressed for brevity). Also s^{-1} is the inverse substitution function in AES.

$$\begin{aligned} x_0 = {} & 0e \bullet s^{-1}(c_0 \oplus k_0) \oplus 0b \bullet s^{-1}(c_{13} \oplus k_{13}) \\ & \oplus 0d \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus 09 \bullet s^{-1}(c_7 \oplus k_7) \\ & \oplus 0e \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \oplus 0b \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ & \oplus 0d \bullet (k_2 \oplus s(k_{11} \oplus k_{15})) \oplus 09 \bullet (k_3 \oplus s(k_8 \oplus k_{12})), \end{aligned} \tag{9}$$

$$\begin{aligned} x_1 = {} & 09 \bullet s^{-1}(c_0 \oplus k_0) \oplus 0e \bullet s^{-1}(c_{13} \oplus k_{13}) \\ & \oplus 0b \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus 0d \bullet s^{-1}(c_7 \oplus k_7) \\ & \oplus 09 \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \oplus 0e \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ & \oplus 0b \bullet (k_2 \oplus s(k_{11} \oplus k_{15})) \oplus 0d \bullet (k_3 \oplus s(k_8 \oplus k_{12})), \end{aligned} \tag{10}$$

$$\begin{aligned} x_2 = {} & 0d \bullet s^{-1}(c_0 \oplus k_0) \oplus 09 \bullet s^{-1}(c_{13} \oplus k_{13}) \\ & \oplus 0e \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus 0b \bullet s^{-1}(c_7 \oplus k_7) \\ & \oplus 0d \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \\ & \oplus 09 \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ & \oplus 0e \bullet (k_2 \oplus s(k_{11} \oplus k_{15})) \oplus 0b \bullet (k_3 \oplus s(k_8 \oplus k_{12})), \end{aligned} \tag{11}$$

$$\begin{aligned} x_3 = {} & 0b \bullet s^{-1}(c_0 \oplus k_0) \oplus 0d \bullet s^{-1}(c_{13} \oplus k_{13}) \\ & \oplus 09 \bullet s^{-1}(c_{10} \oplus k_{10}) \oplus 0e \bullet s^{-1}(c_7 \oplus k_7) \\ & \oplus 0b \bullet (k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36) \\ & \oplus 0d \bullet (k_1 \oplus s(k_{10} \oplus k_{14})) \\ & \oplus 09 \bullet (k_2 \oplus s(k_{11} \oplus k_{15})) \oplus 0e \bullet (k_3 \oplus s(k_8 \oplus k_{12})), \end{aligned} \tag{12}$$

$$\begin{aligned} x_4 = {} & 0e \bullet s^{-1}(c_4 \oplus k_4) \oplus 0b \bullet s^{-1}(c_1 \oplus k_1) \\ & \oplus 0d \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 09 \bullet s^{-1}(c_{11} \oplus k_{11}) \\ & \oplus 0e \bullet (k_0 \oplus k_4) \oplus 0b \bullet (k_1 \oplus k_5) \oplus 0d \bullet (k_2 \oplus k_6) \\ & \oplus 09 \bullet (k_3 \oplus k_7), \end{aligned} \tag{13}$$

$$\begin{aligned} x_5 = {} & 09 \bullet s^{-1}(c_4 \oplus k_4) \oplus 0e \bullet s^{-1}(c_1 \oplus k_1) \\ & \oplus 0b \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 0d \bullet s^{-1}(c_{11} \oplus k_{11}) \\ & \oplus 09 \bullet (k_0 \oplus k_4) \oplus 0e \bullet (k_1 \oplus k_5) \oplus 0b \bullet (k_2 \oplus k_6) \\ & \oplus 0d \bullet (k_3 \oplus k_7), \end{aligned} \tag{14}$$

$$\begin{aligned} x_6 = {} & 0d \bullet s^{-1}(c_4 \oplus k_4) \oplus 09 \bullet s^{-1}(c_1 \oplus k_1) \\ & \oplus 0e \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 0b \bullet s^{-1}(c_{11} \oplus k_{11}) \\ & \oplus 0d \bullet (k_0 \oplus k_4) \oplus 09 \bullet (k_1 \oplus k_5) \oplus 0e \bullet (k_2 \oplus k_6) \\ & \oplus 0b \bullet (k_3 \oplus k_7), \end{aligned} \tag{15}$$

$$\begin{aligned} x_7 = {} & 0b \bullet s^{-1}(c_4 \oplus k_4) \oplus 0d \bullet s^{-1}(c_1 \oplus k_1) \\ & \oplus 09 \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 0e \bullet s^{-1}(c_{11} \oplus k_{11}) \\ & \oplus 0b \bullet (k_0 \oplus k_4) \oplus 0d \bullet (k_1 \oplus k_5) \oplus 09 \bullet (k_2 \oplus k_6) \\ & \oplus 0e \bullet (k_3 \oplus k_7), \end{aligned} \tag{16}$$

$$\begin{aligned} x_8 = {} & 0e \bullet s^{-1}(c_8 \oplus k_8) \oplus 0b \bullet s^{-1}(c_5 \oplus k_5) \\ & \oplus 0d \bullet s^{-1}(c_2 \oplus k_2) \oplus 09 \bullet s^{-1}(c_{15} \oplus k_{15}) \\ & \oplus 0e \bullet (k_4 \oplus k_8) \oplus 0b \bullet (k_5 \oplus k_9) \oplus 0d \bullet (k_6 \oplus k_{10}) \\ & \oplus 09 \bullet (k_7 \oplus k_{11}), \end{aligned} \tag{17}$$

$$x_9 = 09 \bullet s^{-1}(c_8 \oplus k_8) \oplus 0e \bullet s^{-1}(c_5 \oplus k_5) \oplus 0b \bullet s^{-1}(c_2 \oplus k_2) \oplus 0d \bullet s^{-1}(c_{15} \oplus k_{15}) \oplus 09 \bullet (k_4 \oplus k_8) \oplus 0e \bullet (k_5 \oplus k_9) \oplus 0b \bullet (k_6 \oplus k_{10}) \oplus 0d \bullet (k_7 \oplus k_{11}), \quad (18)$$

$$x_{10} = 0d \bullet s^{-1}(c_8 \oplus k_8) \oplus 09 \bullet s^{-1}(c_5 \oplus k_5) \oplus 0e \bullet s^{-1}(c_2 \oplus k_2) \oplus 0b \bullet s^{-1}(c_{15} \oplus k_{15}) \oplus 0d \bullet (k_4 \oplus k_8) \oplus 09 \bullet (k_5 \oplus k_9) \oplus 0e \bullet (k_6 \oplus k_{10}) \oplus 0b \bullet (k_7 \oplus k_{11}), \quad (19)$$

$$x_{11} = 0b \bullet s^{-1}(c_8 \oplus k_8) \oplus 0d \bullet s^{-1}(c_5 \oplus k_5) \oplus 09 \bullet s^{-1}(c_2 \oplus k_2) \oplus 0e \bullet s^{-1}(c_{15} \oplus k_{15}) \oplus 0b \bullet (k_4 \oplus k_8) \oplus 0d \bullet (k_5 \oplus k_9) \oplus 09 \bullet (k_6 \oplus k_{10}) \oplus 0e \bullet (k_7 \oplus k_{11}), \quad (20)$$

$$x_{12} = 0e \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus 0b \bullet s^{-1}(c_9 \oplus k_9) \oplus 0d \bullet s^{-1}(c_6 \oplus k_6) \oplus 09 \bullet s^{-1}(c_3 \oplus k_3) \oplus 0e \bullet (k_8 \oplus k_{12}) \oplus 0b \bullet (k_9 \oplus k_{13}) \oplus 0d \bullet (k_{10} \oplus k_{14}) \oplus 09 \bullet (k_{11} \oplus k_{15}), \quad (21)$$

$$x_{13} = 09 \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus 0e \bullet s^{-1}(c_9 \oplus k_9) \oplus 0b \bullet s^{-1}(c_6 \oplus k_6) \oplus 0d \bullet s^{-1}(c_3 \oplus k_3) \oplus 09 \bullet (k_8 \oplus k_{12}) \oplus 0e \bullet (k_9 \oplus k_{13}) \oplus 0b \bullet (k_{10} \oplus k_{14}) \oplus 0d \bullet (k_{11} \oplus k_{15}), \quad (22)$$

$$x_{14} = 0d \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus 09 \bullet s^{-1}(c_9 \oplus k_9) \oplus 0e \bullet s^{-1}(c_6 \oplus k_6) \oplus 0b \bullet s^{-1}(c_3 \oplus k_3) \oplus 0d \bullet (k_8 \oplus k_{12}) \oplus 09 \bullet (k_9 \oplus k_{13}) \oplus 0e \bullet (k_{10} \oplus k_{14}) \oplus 0b \bullet (k_{11} \oplus k_{15}), \quad (23)$$

$$x_{15} = 0b \bullet s^{-1}(c_{12} \oplus k_{12}) \oplus 0d \bullet s^{-1}(c_9 \oplus k_9) \oplus 09 \bullet s^{-1}(c_6 \oplus k_6) \oplus 0e \bullet s^{-1}(c_3 \oplus k_3) \oplus 0b \bullet (k_8 \oplus k_{12}) \oplus 0d \bullet (k_9 \oplus k_{13}) \oplus 09 \bullet (k_{10} \oplus k_{14}) \oplus 0e \bullet (k_{11} \oplus k_{15}). \quad (24)$$

For ease of explanation, we refer to (9)–(12) as Set-1 equations, (13)–(16) as Set-2, (17)–(20) as Set-3 and (21)–(24) as Set-4 equations. Consider the equations in Set-2, Set-3 and Set-4. Because field multiplication is distributive over field addition, it is possible to split each of the last four terms in the RHS of those equations. Upon rearranging terms, (14), for example, can be re-written as

$$\begin{aligned} x_5 \oplus\ & 09 \bullet s^{-1}(c_4 \oplus k_4) \oplus 0e \bullet s^{-1}(c_1 \oplus k_1) \\ & \oplus\ 0b \bullet s^{-1}(c_{14} \oplus k_{14}) \oplus 0d \bullet s^{-1}(c_{11} \oplus k_{11}) \\ & \oplus\ 09 \bullet ((k_0 \oplus k_4)'0000) \oplus 0e \bullet ((k_1 \oplus k_5)'0000) \\ & \oplus\ 0b \bullet ((k_2 \oplus k_6)'0000) \oplus\ 0d \bullet ((k_3 \oplus k_7)'0000) \\ = {} & 09 \bullet (0000\,(k_0 \oplus k_4)'') \oplus 0e \bullet (0000\,(k_1 \oplus k_5)'') \\ & \oplus\ 0b \bullet (0000\,(k_2 \oplus k_6)'') \oplus\ 0d \bullet (0000\,(k_3 \oplus k_7)''). \end{aligned} \quad (25)$$

Let the RHS of (25) be equal to the byte denoted as $(u_0\,u_1\,u_2\,u_3\ u_4\,u_5\,u_6\,u_7)$. Also, let $a_0a_1a_2z_0$, $a_3a_4a_5z_1$, $a_6a_7a_8z_2$ and $a_9a_{10}a_{11}z_3$ denote the 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.

Multiplication in a binary field involves shifting and XORing of the multiplicand and, if necessary, reduction of the result by an irreducible polynomial. Table 3 shows the shift operation on RHS terms of (25). The high-order nibble (u_0, u_1, u_2, u_3) of the sum of the terms on the RHS is

$$\begin{aligned} u_0 &= 0, \\ u_1 &= a_0 \oplus a_3 \oplus a_6 \oplus a_9, \\ u_2 &= a_1 \oplus a_3 \oplus a_4 \oplus a_7 \oplus a_9 \oplus a_{10}, \\ u_3 &= a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_8 \oplus a_{10} \oplus a_{11}. \end{aligned}$$

Table 3. Field multiplications involved in RHS of (25).

u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	
–	a_0	a_1	a_2	z_0	–	–	–	9 (1001)
–	–	–	–	a_0	a_1	a_2	z_0	
–	a_3	a_4	a_5	z_1	–	–	–	e (1110)
–	–	a_3	a_4	a_5	z_1	–	–	
–	–	–	a_3	a_4	a_5	z_1	–	
–	a_6	a_7	a_8	z_2	–	–	–	b (1011)
–	–	–	a_6	a_7	a_8	z_2	–	
–	–	–	–	a_6	a_7	a_8	z_2	
–	a_9	a_{10}	a_{11}	z_3	–	–	–	d (1101)
–	–	a_9	a_{10}	a_{11}	z_3	–	–	
–	–	–	–	a_9	a_{10}	a_{11}	z_3	

From table 3, it is evident that the bits denoted z_0, z_1, z_2 and z_3 do not affect the high-order nibble of the resultant byte. It follows that, of the 16 unknown bits corresponding to the four terms on the RHS of (25), only 12 bits determine the high-order nibble on the RHS.

The operations involved in calculating u_1, u_2 and u_3 can be represented using the matrix equation

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = M_{9ebd} \bullet A \tag{26}$$

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 coefficients in the RHS of (25).

Let $\mathbb{F}_2^{12}$ represent a 12-dimensional binary vector space. We define 8 equivalence classes as follows:

$$C_{9ebd}\left((u_1, u_2, u_3)^T\right) = \left\{ A \in \mathbb{F}_2^{12} : M_{9ebd} \bullet A = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \right\}. \tag{27}$$

M_{9ebd} is in row canonical form with pivots in the first 3 columns. Hence, a_0, a_1, a_2 are pivot variables and the remaining nine are free variables in A. Thus, each equivalence class has $2^9 = 512$ sub-keys with class representative $(u_1u_2u_3\,000\,000\,000)$. It leads to the following theorem.

Theorem 2 *$\mathbb{F}_2^{12}$ can be partitioned into 8 equivalence classes based on (27), each containing 512 sub-keys. The class representatives are* $(\{0,1\}^3\,000\,000\,000)$.

This 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 in the RHS of (13), (15) and (16) are shifted versions of those in (14). Analogous to M_{9ebd}, we define M_{ebd9}, M_{bd9e} and M_{d9eb}. These matrices are obtained in a manner similar to M_{9ebd} (table 3) and are column-shifted versions of M_{9ebd}. Moreover, they are linearly related as follows:

$$M_{ebd9} = M_1 \bullet M_{9ebd} \tag{28}$$

$$M_{d9eb} = M_2 \bullet M_{9ebd} \tag{29}$$

$$M_{bd9e} = M_3 \bullet M_{9ebd} \tag{30}$$

where $\mathrm{M}_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$, $\mathrm{M}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$ and $M_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$. (31)

Let $A_1 \in C_{9ebd}\left((u_1, u_2, u_3)^T\right)$. From (27)

$$M_{9ebd} \bullet A_1 = (u_1, u_2, u_3)^T.$$

Pre-multiplying by M_1 on both sides gives

$$M_{ebd9} \bullet A_1 = M_1 \bullet (u_1, u_2, u_3)^T.$$

Hence, $A_1 \in C_{ebd9}\left(M_1 \bullet (u_1, u_2, u_3)^T\right)$. This leads to the following theorem.

Theorem 3 *If $A_1 \in C_{9ebd}\left((u_1, u_2, u_3)^T\right)$, then*

$$A_1 \in C_{ebd9}\left(M_1 \bullet (u_1, u_2, u_3)^T\right),$$
$$A_1 \in C_{bd9e}\left(M_2 \bullet (u_1, u_2, u_3)^T\right),$$
$$A_1 \in C_{d9eb}\left(M_3 \bullet (u_1, u_2, u_3)^T\right).$$

The following is of crucial importance in the design of Algorithm 2.

Corollary 1 *If A_1, A_2 belong to an 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 $\mathbb{F}_2^{12}$.

5.2 *Algorithm 2: description*

The guesstimate sets that are inputs to Algorithm 2 are derived as follows. Let r be the maximum number of runs, from the starting run of decryption d containing no more than 8 distinct cache line numbers. Let n be the minimum integer such that the number of distinct line numbers in runs $r+1, r+2, \ldots, r+n$ is 8 or more. Let γ be the set containing these line numbers. Then populate $G_{i,t,2,d}$, $0 \le i, t \le 3$, with line numbers from γ in the range from $16t$ to $16(t+1)-1$. From the guesstimate sets and ciphertext, we derived the four histograms using Algorithm 2.

Algorithm 2 builds three histograms corresponding to the three sets of equations (Set-2, Set-3 and Set-4). A histogram contains 2^{19} bins, each representing a possible sub-key. A sub-key for Set-2, for example, is formed by concatenating $k''_4, k''_1, k''_{14}, k''_{11}$ to a three-bit attribute (lines 14–16 of algorithm 2). The first four nibbles are the unknowns in the LHS of Eq. (25).

Algorithm 2: Second Round Attack

Input: Ciphertext, $c_{t+4i,d}$ and Guesstimates, $G_{i,t,2,d}$ $0 \le i \le 3$, $0 \le t \le 3$ and $1 \le d \le \delta$

Output: Low-order nibble of each byte of AES key

r_1 is set of all binary strings of length 16
r_2, r_3, r_4 are sets of all binary strings of length 19
for *each* r_a, $2 \le a \le 4$ **do**
  for *each decryption* d, $1 \le d \le \delta$ **do**
    for *each* n, $1 \le n \le 4$ **do**
      for *each* $j \in r_a$ **do**
        if *j satisfies Eq. No.* $(n + 4a)$ **then**
          increment $hist_{r_a}[j]$
        end
      end
    end
  end
end
$\left(k''_4, k''_1, k''_{14}, k''_{11}, y_1\right) = argmax_j hist_{r_2}[j]$
$\left(k''_8, k''_5, k''_2, k''_{15}, y_2\right) = argmax_j hist_{r_3}[j]$
$\left(k''_{12}, k''_9, k''_6, k''_3, y_3\right) = argmax_j hist_{r_4}[j]$
for *each decryption* d, $1 \le d \le \delta$ **do**
  for *each* n, $1 \le n \le 4$ **do**
    for *each* $j \in r_1$ **do**
      if *j satisfies Eq. No.* $(n + 4)$ **then**
        increment $hist_{r_1}[j]$
      end
    end
  end
end
$\left(k''_0, k''_{13}, k''_{10}, k''_7\right) = argmax_j hist_{r_1}[j]$

From Theorem 2, the space of unknown key bits in the RHS of Eq. (25) that impacts the high-order nibble of the RHS can be reduced to just eight equivalence classes, each represented by three bits. These three bits comprise the last attribute of the sub-key. Further, from Corollary 1, the equivalence classes and their representatives are identical across the four equations of a set. Hence, the same "sub-key" value participates in four Bernoulli trials per decryption. As in the Round 1 Attack, a trial involves computing the high-order nibble of the byte value of the RHS and checking to see if it matches an element in the corresponding set of guesstimates. If so, the score of that sub-key value is incremented by 1 in the histogram. After δ decryptions, we pick the sub-key with the highest score and declare it to be the winner.

The afore-mentioned procedure is completed for the three histograms and (hopefully) retrieves 12 low-order nibbles of the key. To recover the remaining nibbles, we create a histogram to score each of the 2^{16} values of the sub-key made up of $k''_0, k''_{13}, k''_{10}, k''_7$. We use the first set of equations (Set-1) to do so. Once again, we have 4δ Bernoulli trials but now only 2^{16} possible sub-key values (against 2^{19} in the earlier three histograms). The winner in this histogram is presumed to yield the true values of $k''_0, k''_{13}, k''_{10}, k''_7$.

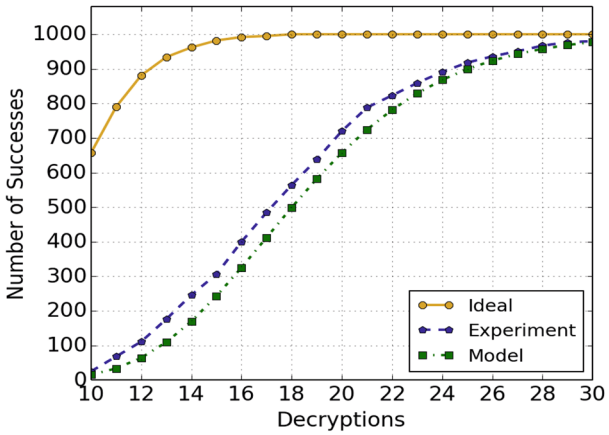

Figure 3. Number of successes per 1000 samples (Round 2).

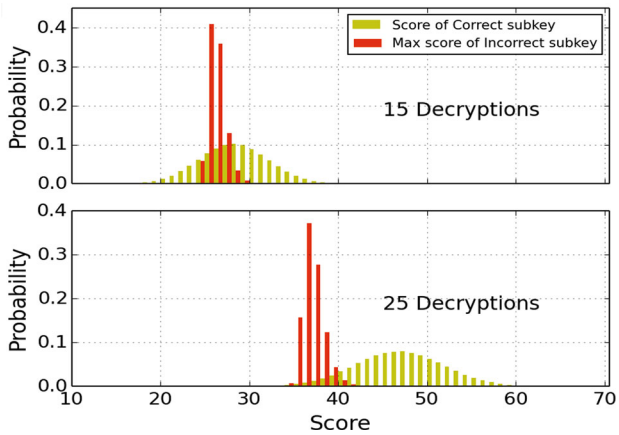

Figure 4. Distribution of the correct sub-key score (yellowish green) and distribution of the maximum score among the incorrect sub-keys (red).

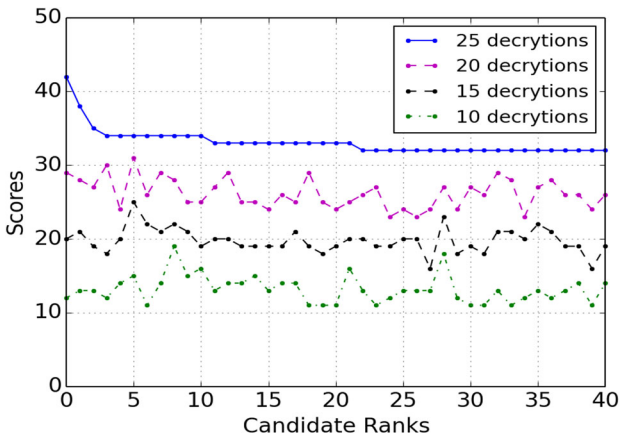

Figure 5. Scores of top 40 sub-keys after varying number of decryptions

5.3 *Analysis and results*

As in section 4, f_0 and $|G|_0$ are, respectively, the false negative rate and the average cardinality of the set of guesstimates. Also p_c and p_{in} are, respectively, the probabilities of the correct and incorrect sub-keys receiving a boost. Hence

$$p_c = 1 - f_0 \tag{32}$$

and

$$p_{in} = \frac{|G|_0}{16}. \tag{33}$$

From Lemma 1, we obtain probabilities of successfully recovering the sub-keys from the four histograms. Combining the four probabilities, we obtain the following theorem.

Theorem 4 *The probability of successfully recovering all 16 low-order nibbles of the AES key in δ decryptions is*

$$\left[P_h\left(1 - f_0, \frac{|G|_0}{16}, 2^{19}, 4\delta\right)\right]^3 P_h\left(1 - f_0, \frac{|G|_0}{16}, 2^{16}, 4\delta\right)$$

For the Second Round Attack, we used the same 1000 samples and spy input as in the First Round Attack. Figure 3 shows the number of unqualified successes per 1000 samples for varying number of decryptions. In these cases, the four histograms throw up unique winners, each with at least a four-point lead over their nearest rivals. If the four winners collectively yield the correct key, we refer to the experiment as an unqualified success.

Theoretically, the distribution of the score of correct sub-key and the distribution of the maximum score among 2^{19} incorrect sub-keys' scores are shown in figure 4 after 15 and 25 decryptions. To calculate these distributions, we assumed $f_0 = 0.53$ and $|G|_0 = 2.7$ based on our experiments. In our model, scores of correct and incorrect sub-keys are binomially distributed with success probabilities p_c and p_{in}, respectively, which can be calculated using Eqs. (32) and (33). It is also interesting to note that while the distribution of scores of the correct sub-key value has no skew or excess kurtosis, the distribution of the maximum score among incorrect sub-keys' scores exhibits positive skew and positive excess kurtosis. Using our model we estimated that the probability of the score of the correct sub-key exceeding the highest score among incorrect sub-keys is, respectively, 0.66, 0.88 and 0.97 for 15, 20 and 25 decryptions.

Figure 5 shows the scores of the 40 top-ranked sub-keys in a histogram. The sub-keys are arranged in order of their scores obtained after 25 decryptions. Their scores after 10, 15 and 20 decryptions are also shown. The correct sub-key emerges as the clear winner after only about 22 decryptions but has 6, 20 and 26 sub-keys at the same or higher score after 20, 15 and 10 decryptions, respectively. Hence, to guess the true sub-key value more accurately, we make the following slight modification to Algorithm 2.

If the top scorer in a histogram does not have a lead of at least 4 points over the rest, we identify all sub-key values with scores greater than or equal to the minimum score of the top eight sub-key values. We then construct a histogram for each combination of these top scorers from the first three histograms. Scores for the same 2^{16} sub-key values in each histogram are obtained as before from the Set-1 equations. The final winner for the complete AES key is obtained by selecting the global top scorer over all histograms constructed in the final step. With this modification, the success rate for 25 decryptions, for example, increased from 92% to 96%.

The experimentally obtained average values of f_0 and $|G|_0$ for the 1000 samples were, respectively, 0.53 and 2.71. $|G|_0$ in the First Round Attack was much lower at 2.15. This is because the first run of a decryption can be unambiguously identified and contains only accesses made in Round 1. This is not true for the first run of Round 2. From Eqs. (32) and (33), the success probabilities p_c and p_{in} are, respectively, 0.47 and 0.17. The means of the distributions of the correct and incorrect sub-key scores (equal to $4\delta p_c$ and $4\delta p_{in}$) are much further apart than in the First Round Attack (equal to δp_c and δp_{in}). This might suggest that the winner in the Second Round Attack can be identified with fewer decryptions.

The number of sub-key values in the histogram for Round 1 is only 16 while it is 2^{16} or 2^{19} for Round 2. Thus, despite smaller intersection of the tails of the two distributions of Round 2, the sheer number of sub-key values increases the probability that one of the incorrect sub-key values has a higher score than the actual sub-key[2]. These two effects seem to balance out, resulting in roughly the same number of decryptions required for the First and Second Round Attacks.

Figure 3 also shows the successes in the case of ideal guesstimates. The success rate is nearly 100% with just 15 decryptions. This is attributed to the fact that $|G|_0$ in the ideal case is only 1.82 (average over 1000 samples).

6. Discussion

In this section, we discuss further optimizations along with limitations of our approach and countermeasures.

6.1 *Further optimizations and enhancements*

With prefetching disabled, we were able to successfully retrieve all 16 bytes of the AES key in 2–3 decryptions.

[2]The higher variance for the Round 2 distribution is a secondary effect, which favours a larger number of decryptions for the Second Round Attack.

From the deletion table, we obtained the near-exact order in which the line numbers were accessed, resulting in about one line number per guesstimate set. With prefetching enabled, our spy code skips the odd-numbered lines and we get only a partial order of the even-numbered lines accessed. However, using the temporal information provided by the deletion table, we could use different per-element guesstimate sets instead of using the same guesstimate set for all elements accessing a given table. Moreover, weights could be assigned to different elements in the guesstimate set, resulting in fractional scores assigned to sub-keys rather than 0/1 scores in the current histograms. We expect that this change will speed up the convergence of our algorithms. Finally, if Algorithm 2 throws up multiple candidates for the final key, we could use equations for the third round to resolve the ties.

Most of the Intel caches have a block size of 64 bytes. However, the IBM Power PC processor has block size of 128 bytes. Our attack can be adapted to work on the latter. The First Round Attack will obtain the first three bits of each AES key. Obtaining the remaining five bits (Second Round Attack) will take a lot more time since the space of sub-keys will be much larger now.

6.2 *Limitations*

We assume that the core running the victim and spy does not simultaneously host another active process running AES decryption. Otherwise, the table accesses made by the latter may be mistaken for accesses by the victim, possibly leading to flawed conclusions.

As with some access-driven attacks, we assume that the victim and spy process are on the same processor core. This is not always possible though a determined attacker may be able to co-locate itself with the victim. For example, in a multiuser environment, an attacker together with her accomplices could simply request inordinate CPU resources and obtain access to multiple cores, including the one victim is running on.

Hardware support for AES is available via the AES-NI instructions on many modern processors beginning with Intel's Westmere family. Since the hardware implementation does not use processor cache to store the look-up tables, the attack described here will not work. Use of the AES-NI instructions (rather than the software implementation) is the default option with the newer OpenSSL versions. 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. The default option in more recent versions of OpenSSL is the assembly language implementation. This uses only a single table and may not be vulnerable to the attacks presented in this paper. However, it can be easily overridden by setting no-asm flag during compilation.

Most recent Linux kernel versions support control groups (cgroups). When enabled, the CPU will be allocated equally among the processes of different cgroups. If the victim and spy are in different cgroups, then the victim will get roughly the same amount of CPU time as the spy, so it will perform many decryptions during its run, thus rendering this attack infeasible.

6.3 *Countermeasures*

An extreme countermeasure is physical isolation between a sensitive application and all others. The more practical countermeasures are either processor-based or OS-based. In [48], the OS makes spurious requests to obfuscate cache usage. Most cache-based side-channel attacks are critically dependent on the ability of the adversary to measure the state of the cache frequently. This is achieved by pre-emption of the victim after a few microseconds. By modifying the OS to enforce a lower limit of say 100 μs on the minimum run time of a CPU-bound process [49], attacks such as these may be thwarted.

A processor-based defense is to drop the flush instruction from the instruction repertoire as in some versions of ARM. However, [50] shows that it may still be possible to evict cache lines by accesses to the same cache set. In this paper, we have retrieved the AES key despite the presence of the hardware prefetcher. Existing prefetchers could be enhanced by randomized and set-balanced stride prefetchers [46] to severely disrupt the cache footprint of a side-channel attack, leading to negligible leakage of useful information.

Removing the support of the high-resolution timer [10, 16] or reducing its accuracy [51, 52] has been suggested as a countermeasure to the cache attacks. However, it has been demonstrated [50] that removal of the timer is not a solution as a very high resolution can be achieved by incrementing a global variable in an endless loop by a counting thread. The boundaries (maximum stride) of the prefetcher can also be extended to defend against this attack.

7. Conclusion

We implemented cache access attacks on the Core 2 Duo and Core i3/i5/i7 processors to recover the AES key using a multi-threaded spy process. Hardware prefetchers on modern machines complicate cache access attacks as the prefetched cache lines are wrongly reported as being accessed by the victim. Our spy code decreased the number of false positives but greatly increased the number of false negatives. Yet, our key retrieval algorithms required about 25 blocks of ciphertext to retrieve the key with prefetching enabled and only 2–3 blocks with

prefetching disabled. We also presented analytical models, which provide deeper insight into the effect of prefetching and of errors (false positives and false negatives) on performance.

Appendix I: Proof of Theorem 1

Say f_q and $\mid G \mid_q$ are, respectively, the average false negative rate and the average cardinality of the set of guesstimates after q refinements are made in initial set of guesstimates. Average false negative rate can also be thought of as the probability of occurrence of a false negative:

$$p_c = 1 - f_q. \tag{A1}$$

This equation follows from the observation that the correct nibble in the histogram receiving a boost and the occurrence of a false negative are mutually exclusive and exhaustive events. To derive p_{in}, consider the following reasoning. When there is a false negative, all the guesstimates lead to boosting some incorrect nibble in the histogram; hence in this case, probability of an incorrect nibble receiving a boost is $\frac{|G|_q}{15}$, since there are 15 incorrect nibbles. When there is no false negative, all the guesstimates except one lead to boosting some incorrect nibble in the histogram. This guesstimate boosts the correct nibble. Hence, in this case, the probability of an incorrect nibble receiving a boost is $\frac{|G|_q - 1}{15}$. Combining both the cases, we obtain the following equation:

$$p_{in} = f_q\left(\frac{\mid G \mid_q}{15}\right) + (1 - f_q)\left(\frac{\mid G \mid_q - 1}{15}\right). \tag{A2}$$

After obtaining a key nibble, we refine the set of guesstimates. The cardinality of the set of guesstimates decreases by 1 with probability $1 - f_q$. This is the probability of absence of a false negative, i.e., there must be a guesstimate that led to boost of correct nibble and it was removed from set of guesstimates during refinement. Also, the average false negative rate increases due to refinement since some line numbers may be accessed more than once in a round and they are removed during refinement:

$$\mid G \mid_{q+1} = \mid G \mid_q - (1 - f_q),\ 0 \le q \le 2. \tag{A3}$$

Let f be the probability of the occurrence of a false negative due to the spy input and preprocessing strategy. Let p_q be the probability of false negative occurrence due to refining of guesstimate sets (some line numbers may be accessed more than once in a round) after q refinements. Assuming that these two sources of false negatives are independent of each other, we have the following equation:

$$f_q = f + p_q - (f \bullet p_q),\ 0 \le q \le 3 \tag{A4}$$

where $p_0 = 0,\ p_1 = \frac{1}{16},\ p_2 = \frac{31}{256},\ p_3 = \frac{721}{4096}$.

As p_0 corresponds to zero refinements, there would not be any false negatives due to refinement, so $p_0 = 0$. After refining once, there is a chance of false negative occurrence since the removed line number might have been accessed more than once. For the second nibble to be recovered, the removed line number is a false negative if the removed line number is accessed due to this nibble. As there are 16 possible line numbers, probability of this event is $\frac{1}{16}$. Hence, $p_1 = \frac{1}{16}$.

After two refinements, the probability of a false negative occurrence due to refinement is equal to the probability of line number accessed being either of the first two nibbles recovered, which is $\frac{16+16-1}{16x16} = \frac{31}{256}$. Hence $p_2 = \frac{31}{256}$.

After three refinements, the probability of false negative occurrence due to refinement is equal to probability of the line number accessed being either of the first three nibbles recovered. Let A_i be the event in which the line number accessed corresponding to i^{th} nibble recovered matches corresponding to last nibble recovered. Then, p_3 is $P(A_1 \cup A_2 \cup A_3)$, which is

$$\sum_{i=1}^{3} P(A_i) - \sum_{i=1}^{3}\sum_{j>i}^{3} P(A_i \cap A_j) + P(A_1 \cap A_2 \cap A_3).$$

When matching with any one of the three, other two are free to take any of the 16 possible values; hence out of $16 \times 16 \times 16$ possibilities, 16×16 are favourable, so $P(A_i) = \frac{256}{4096}$. When matching with any two of the three simultaneously, the third one is free to take any of the 16 possible values. Out of $16 \times 16 \times 16$ possible cases, 16 are favourable, so $P(A_i \cap A_j) = \frac{16}{4096}$. As there is only 1 way in which all three match, out of $16 \times 16 \times 16$ possible cases, $P(A_1 \cap A_2 \cap A_3) = \frac{1}{4096}$. Using these values, we can calculate

$$p_3 = 3 \times \frac{256}{4096} - 3 \times \frac{16}{4096} + \frac{1}{4096} = \frac{721}{4096}.$$

As explained in section 4.3, probability of recovering the first nibble among $k'_{4m}, 0 \le m \le 3$, is

$$1 - [1 - P_h(p_c, p_{in}, 2^4, \delta)]^4.$$

Hence, probability of retrieving a nibble after q refinements is

$$1 - [1 - \mathcal{P}_h(f_q, \mid G \mid_q, 2^4, \delta)]^{4-q}.$$

where $\mathcal{P}_h(f_q, \mid G \mid_q, 2^4, \delta) = P_h(p_c, p_{in}, 2^4, \delta)$.

Hence, probability of retrieving all the 4 nibbles $k'_{4m}, 0 \le m \le 3$, is

$$\prod_{q=0}^{3}\left[1 - [1 - \mathcal{P}_h(f_q, \mid G \mid_q, 2^4, \delta)]^{4-q}\right].$$

It is also the same as the probability of correctly retrieving all the four nibbles $k'_{t+4m}, 0 \le m \le 3$, for a given $t, 0 \le t \le 3$. Hence, overall probability of retrieving all the 16 high-order nibbles is

$$\left\{ \prod_{q=0}^{3} \left[1 - [1 - \mathcal{P}_h(f_q, \mid G \mid_q, 2^4, \delta)]^{4-q} \right] \right\}^4.$$

Appendix II: AES equations and T-table usage

Deriving equations

Input to Round 1 of decryption is

$$\begin{pmatrix} c_0 \oplus k_0 & c_4 \oplus k_4 & c_8 \oplus k_8 & c_{12} \oplus k_{12} \\ c_1 \oplus k_1 & c_5 \oplus k_5 & c_9 \oplus k_9 & c_{13} \oplus k_{13} \\ c_2 \oplus k_2 & c_6 \oplus k_6 & c_{10} \oplus k_{10} & c_{14} \oplus k_{14} \\ c_3 \oplus k_3 & c_7 \oplus k_7 & c_{11} \oplus k_{11} & c_{15} \oplus k_{15} \end{pmatrix}$$

where $C = (c_0, c_1, \ldots, c_{15})$ and $K = (k_0, k_1, \ldots, k_{15})$, respectively, denote ciphertext and tenth round key (in terms of key scheduling algorithm used for encryption; in implementation for storage-constrained environments, this key is stored and other round keys are generated on the fly). After inverse byte substitution and inverse row shift operations, input transforms to

$$\begin{pmatrix} s^{-1}(c_0 \oplus k_0) & s^{-1}(c_4 \oplus k_4) & s^{-1}(c_8 \oplus k_8) & s^{-1}(c_{12} \oplus k_{12}) \\ s^{-1}(c_{13} \oplus k_{13}) & s^{-1}(c_1 \oplus k_1) & s^{-1}(c_5 \oplus k_5) & s^{-1}(c_9 \oplus k_9) \\ s^{-1}(c_{10} \oplus k_{10}) & s^{-1}(c_{14} \oplus k_{14}) & s^{-1}(c_2 \oplus k_2) & s^{-1}(c_6 \oplus k_6) \\ s^{-1}(c_7 \oplus k_7) & s^{-1}(c_{11} \oplus k_{11}) & s^{-1}(c_{15} \oplus k_{15}) & s^{-1}(c_3 \oplus k_3) \end{pmatrix}.$$

For keys generated using key scheduling algorithm for encryption, round key addition and then inverse column mixing should be performed. To have a similar structure to decryption as that of encryption, round key addition and inverse column mixing steps are interchanged but this requires that the round key is suitably transformed. Here, first we will consider doing round key addition and then we perform inverse column mixing.

Let W_{36}, W_{37}, W_{38} and W_{39} denote 4 words (1 word = 4 bytes) of 9th round key in encryption procedure. Let W_{40}, W_{41}, W_{42} and W_{43} denote 4 words of 10th round key in encryption procedure. According to key scheduling algorithm for encryption, these words are related as described in following equations:

$$\begin{aligned} W_{40} &= W_{36} \oplus f(W_{39}), \\ W_{41} &= W_{37} \oplus W_{40}, \\ W_{42} &= W_{38} \oplus W_{41}, \\ W_{43} &= W_{39} \oplus W_{42}. \end{aligned}$$

These equations are used to obtain the 10th round key using the 9th round key in encryption. In these equations, $f(W)$ is obtained by first doing one left cyclic rotation of bytes of word W and then applying S-box on each of the bytes. It is then XORed with a round-dependent constant. We can manipulate these equations to obtain the 9th round key, given the 10th round key:

$$\begin{aligned} W_{36} &= W_{40} \oplus f(W_{39}), \\ W_{37} &= W_{40} \oplus W_{41}, \\ W_{38} &= W_{41} \oplus W_{42}, \\ W_{39} &= W_{42} \oplus W_{43}. \end{aligned}$$

The equation to obtain W_{36} can be re-written as

$$W_{36} = W_{40} \oplus f(W_{42} \oplus W_{43}),$$

using these equations. Combining two different notations for the 10th round key, we have

$$(\, W_{40} \quad W_{41} \quad W_{42} \quad W_{43} \,) = \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}.$$

According to the definition of $f(W)$

$$f(W_{42} \oplus W_{43}) = \begin{pmatrix} s(k_9 \oplus k_{13}) \oplus 36 \\ s(k_{10} \oplus k_{14}) \\ s(k_{11} \oplus k_{15}) \\ s(k_8 \oplus k_{12}) \end{pmatrix}$$

where 36 is the round-dependent constant. In any round-dependent constant, last three bytes are all zeros. Hence, the 9th round key of encryption is

$$\begin{pmatrix} k_0 \oplus s(k_9 \oplus k_{13}) \oplus 36 & k_0 \oplus k_4 & k_4 \oplus k_8 & k_8 \oplus k_{12} \\ k_1 \oplus s(k_{10} \oplus k_{14}) & k_1 \oplus k_5 & k_5 \oplus k_9 & k_9 \oplus k_{13} \\ k_2 \oplus s(k_{11} \oplus k_{15}) & k_2 \oplus k_6 & k_6 \oplus k_{10} & k_{10} \oplus k_{14} \\ k_3 \oplus s(k_8 \oplus k_{12}) & k_3 \oplus k_7 & k_7 \oplus k_{11} & k_{11} \oplus k_{15} \end{pmatrix}.$$

We XOR this matrix and the result we obtain after inverse byte substitution and inverse row shift operations. Next, we perform inverse column mixing to obtain the output of first round of decryption:

$$B^{-1} = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix}.$$

Hence, the last step before obtaining output of first round of decryption is pre-multiplication by this matrix B^{-1}.

Appendix II.1: explaining T-tables

Each $T_t, 0 \le t \le 3$, takes one byte as input and returns 4 output bytes. Output from T_t table is product of t^{th} column

of B^{-1} and inverse S-Box applied to the input of the table. Hence

$$T_0[\alpha] = (0e \bullet s^{-1}(\alpha), 09 \bullet s^{-1}(\alpha), 0d \bullet s^{-1}(\alpha), 0b \bullet s^{-1}(\alpha)),$$
$$T_1[\alpha] = (0b \bullet s^{-1}(\alpha), 0e \bullet s^{-1}(\alpha), 09 \bullet s^{-1}(\alpha), 0d \bullet s^{-1}(\alpha)),$$
$$T_2[\alpha] = (0d \bullet s^{-1}(\alpha), 0b \bullet s^{-1}(\alpha), 0e \bullet s^{-1}(\alpha), 09 \bullet s^{-1}(\alpha)),$$
$$T_3[\alpha] = (09 \bullet s^{-1}(\alpha), 0d \bullet s^{-1}(\alpha), 0b \bullet s^{-1}(\alpha), 0e \bullet s^{-1}(\alpha)).$$

Total possible number of inputs is 2^8. A table stores 4 bytes at each index position, so the size of each table size is $4 \times 2^8 = 2^{10}$ bytes = 1 KB.

References

[1] Zhou Y B and Feng D 2005 Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptol. ePrint Arch.*: 388

[2] Joan D and Rijmen V 2002 *The Design of Rijndael: AES—The Advanced Encryption Standard*. Springer

[3] OpenSSL Software Foundation. Openssl project. https://www.openssl.org/. Accessed Apr 2018

[4] Hennessy J L and Patterson D A 2011 *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. 5th edn., Morgan Kaufmann, Burlington. p. 9

[5] Yarom Y and Falkner K 2014 Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, USENIX Association, pp. 719–732

[6] Ashokkumar C, Giri R P and Menezes B March 2016 Highly efficient algorithms for AES key retrieval in cache access attacks. In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 261–275

[7] Ashokkumar C, Venkatesh M B S, Giri R P and Menezes B 2016 Design, Implementation and performance analysis of highly efficient algorithms for AES key retrieval in access-driven cache-based side channel attacks. Technical report, Department of Computer Science and Engineering, IIT-Bombay

[8] Eran T, DagArne O and Adi S 2010 Efficient cache attacks on AES and countermeasures. *J. Cryptol.* 23(1): 37–71

[9] Intel Corporation 2016 Intel® 64 and IA-32 Architectures Optimization Reference Manual. Number 248966-033

[10] Gullasch D, Bangerter E and Krenn S 2011 Cache games—bringing access-based cache attacks on AES to practice. In: *IEEE Computer Society, Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11*, Washington, DC, USA, pp. 490–505

[11] Intel Corporation 2017 Intel® 64 and IA-32 Architectures Software Developer's Manual. Number 325462-062US

[12] Hu W-M 1992 Lattice scheduling and covert channels. In: *IEEE Computer Society, Proceedings of the IEEE Symposium on Security and Privacy, SP '92*, Washington, DC, USA, pp. 52–61

[13] Kocher P C 1996 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, CRYPTO '96*, Springer, London, UK, pp. 104–113

[14] John K, Bruce S, David W and Chris H 2000 Side channel cryptanalysis of product ciphers. *J. Comput. Secur.* 8: 141–158

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

[16] Percival C 2005 Cache missing for fun and profit

[17] Neve M and Seifert J-P 2007 Advances on access-driven cache attacks on AES. In: Biham E and Youssef AMRM, editors, *Selected Areas in Cryptography*, *Volume 4356 of Lecture Notes in Computer Science*, Springer, New York, pp. 147–162

[18] Acıiçmez O, Brumley B B and Grabher P 2010 New results on instruction cache attacks. In: *Cryptographic Hardware and Embedded Systems, CHES 2010*, Springer, New York, pp. 110–124

[19] Spreitzer R and Thomas PLOS 2013 Cache-access pattern attack on disaligned AES T-Tables. In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*, Springer, New York, pp. 200–214

[20] Acıiçmez O and Koç Çetin K 2006 Trace-driven cache attacks on AES. In: *Proceedings of the 8th International Conference on Information and Communications Security, ICICS'06*, Springer, Berlin, pp. 112–121

[21] Gallais J F, Kizhvatov I and Tunstall M 2011 Improved trace-driven cache-collision attacks against embedded AES implementations. In: *Information Security Applications*, Springer, New York, pp. 243–257

[22] Zhao X J and Wang T 2010 Improved cache trace attack on AES and CLEFIA by considering cache miss and S-box misalignment. *IACR Cryptology ePrint Archive*, p. 56

[23] Tsunoo Y, Saito T, Suzaki T, Shigeri M and Miyauchi H 2003 Cryptanalysis of DES implemented on computers with cache. In: *Cryptographic Hardware and Embedded Systems—CHES 2003*, *Volume 2779 of Lecture Notes in Computer Science*, Springer, New York, pp. 62–76

[24] Bonneau J and Mironov I 2006 Cache-collision timing attacks against AES. In: Goubin L and Matsui M editors, *Cryptographic Hardware and Embedded Systems—CHES 2006*, *Volume 4249 of Lecture Notes in Computer Science*, Springer, New York, pp. 201–215

[25] Bernstein D J 2005 Cache-timing attacks on AES

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

[27] Neve M, Seifert J-P and Wang Z 2006 A refined look at Bernstein's AES side-channel analysis. In: *Proceedings of the ACM Symposium on Information, computer and communications security*, ACM, pp. 369–369

[28] Osvik D, Shamir A and Tromer E 2006 Cache attacks and countermeasures: The case of AES. In David P, editor, *Topics in Cryptology CT-RSA 2006*, *Volume 3860 of Lecture Notes in Computer Science*, Springer, New York, pp. 1–20

[29] Acıiçmez O, Schindler W and Koç Ç K 2006 Cache based remote timing attack on the AES. In: *Topics in Cryptology—CT-RSA 2007*, Springer, New York, pp. 271–286

[30] Canteaut A, Lauradoux C and Seznec A 2006 Understanding cache attacks. In: *Research Report RR-5881*

[31] Tiri K, Acıiçmez O, Neve M and Andersen F 2007 An analytical model for time-driven cache attacks. In: *Fast Software Encryption*, Springer, New York, pp. 399–413

[32] Rebeiro C, Mondal M and Mukhopadhyay D 2010 Pinpointing cache timing attacks on AES. In: *23rd International Conference on VLSI Design*, IEEE, pp. 306–311

[33] Atici A C, Yilmaz C and Savas E 2013 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, pp. 74–83

[34] S\$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES, author=Irazoqui, Gorka and Eisenbarth, Thomas and Sunar, Berk, booktitle=IEEE Symposium on Security and Privacy, pages=591–604, year=2015, organization=IEEE

[35] Liu F, Yarom Y, Ge Q, Heiser G and Lee R B 2015 Last-level cache side-channel attacks are practical. In: *IEEE Symposium on Security and Privacy*, pp. 605–622

[36] Kayaalp M, Abu-Ghazaleh N, Ponomarev D and Jaleel A 2016 A high-resolution side-channel attack on last-level cache. In: *Proceedings of the 53rd Annual Design Automation Conference*, ACM, p. 72

[37] Zhang Y, Juels A, Reiter M K and Ristenpart T 2012 Cross-VM side channels and their use to extract private keys. In: *Proceedings of the ACM Conference on Computer and Communications Security*, ACM, pp. 305–316

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

[39] Weiß M, Heinz B and Stumpf F 2012 A cache timing attack on AES in virtualization environments. In: *Financial Cryptography and Data Security*, Springer, New York, pp. 314–328

[40] Apacechea G I, Inci M S, Eisenbarth T and Sunar B 2014 Fine grain cross-VM attacks on XEN and VMware are possible! *IACR Cryptology ePrint Archive*, p. 248

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

[42] Baer J-L and Chen T-F 1991 An effective on-chip preloading scheme to reduce data access penalty. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, 1991. Supercomputing'91*, IEEE, pp. 176–186

[43] Yarom Y and Benger N 2014 Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, p. 140

[44] Liu F, Yarom Y, Ge Q, Heiser G and Lee R B 2015 Last-level cache side-channel attacks are practical. In: *IEEE Computer Society Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15* Washington, DC, USA, pp. 605–622

[45] Gruss D, Maurice C, Fogh A, Lipp M and Mangard S 2016 Prefetch side-channel attacks: Bypassing smap and kernel aslr. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, ACM, New York, NY, USA, pp. 368–379

[46] Fuchs A and Lee R B 2015 Disruptive prefetching: Impact on side-channel attacks and cache designs. In: *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR '15*, ACM, New York, NY, USA, pp. 14:1–14:12

[47] Mowery K, Keelveedhi S and Shacham H 2012 Are AES x86 cache timing attacks still feasible? In: *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, ACM, pp. 19–24

[48] Zhang Y and Reiter M K 2013 Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, ACM, New York, NY, USA, pp. 827–838

[49] Varadarajan V, Ristenpart T and Swift M 2014 Scheduler-based defenses against cross-vm side-channels. In: *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, pp. 687–702

[50] Lipp M, Gruss D, Spreitzer R, Maurice C and Mangard S 2016 Armageddon: Cache attacks on mobile devices. In: *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, Austin, TX, pp. 549–564

[51] Hu WM 1992 Reducing timing channels with fuzzy time. *J. Comput. Secur.* 1(3-4):233–254

[52] Vattikonda B C, Das S and Shacham H 2011 Eliminating fine grained timers in XEN. In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ACM, pp. 41–46