Cache Timing Attacks against AES Encryption on x86 and ARM Architectures

Toyin Yusuf, Nagulan Manivelan, Pragadeesh Nithyakumar

Abstract

This paper demonstrates two cache timing side-channel attacks against AES encryption on two architectures. The caches of the x86 and ARM processors can leak information if data is shared between processors. Attackers can get this information by timing how long a cache handles a hit or a miss after the victim does an encryption. We run an AES encryption provided by the OpenSSL library on these architectures, and implement the *Flush+Reload* and *Flush+Flush* timing attacks to retrieve secret encryption keys. These attacks determine if data are cached at a target address. *Flush+Reload* collects timing samples of the flush and reload instructions, while *Flush+Flush* only collects the execution time of the flush instruction. When the cached data is found, an inverse encryption algorithm is used to determine the private encryption key.

1. Introduction

Data dependency is highly present in hardware, as it optimizes throughput in software execution. The CPU cache is a hardware component that is used to decrease the time it takes to access data in memory. It is shared among other hardware components in order to improve performance and decrease power consumption of the system. Because of shared memory lines in the cache, an attacker can setup a shared resource so that it can check whether a victim task has used a specific cache block. If the victim have accessed the shared resource the attacker can retrieve information used by the victim at this resource. During a memory access request, a cache hit, the requested data is present, or miss, requested data is not present, can occur. During a miss, the cache block is replaced by the requested block, resulting in a longer access time. The attacker can exploit this process by timing how long it takes to access this data in memory (i.e. a timing side-channel). Because it takes longer to handle a cache miss, the attacker can determine when a cache hit and miss occurs which therefore determines if the victim has accessed the cache.

This paper will demonstrate cache-timing attacks on two different architectures. These attacks will retrieve private encryption key information after an AES encryption is done on the systems. The rest of the paper is organized into the following sections: Background, Experiments, Results, and Conclusion. The Background section provides information about the AES cipher algorithm and explains the *Flush+Reload* and *Flush+Flush* attack procedures. The Experiments section describes the AES cipher and timing attacks procedure on the x86 and ARM architectures. The Results sections show the cache miss and hit times of the architectures and the partially recovered encryption key used in the encryption. The Conclusion section summarizes our experiments, the results of the side-channel attacks, and further steps to verify the accuracy of the recovered key.

2. Background

2.1 AES Encryption

The AES (Advanced Encryption Standard) cypher is an iterated cipher that was designed by Vincent Rijmen and Joan Daemen. The cipher consists of 10, 12, or 14 rounds, depending on the size of the encryption key. This paper uses a 128-bit key and performs a 10-round encryption. The encryption takes in a 16-byte plaintext and a 16-byte key and returns a 16-byte ciphertext. The ten rounds consists of four operations: byte substitution, shift row, mix columns, and add subkey. After the rounds, an exclusive-or is done with the round key. Intermediate values are stored in lookup T-tables; which gets cached during the encryption. These rounds may demand on the encryption key and the inverse of these rounds can be used on the ciphertext to gain back the plaintext. This experiment uses the inverse of the sbox, used in the last round of AES, to recover the key. AES is included in the OpenSSL software library, which provide secure communication and cryptography applications.

2.2 Flush+Reload Attack

Many cache side-channel attacks have been developed to exploit CPU cache memory sharing behavior. One of these is the *Flush+Reload* attack. This attack targets the Last-Level Cache (LLC), which is shared between the victim and attacker processes. The attacker can monitor accesses to the cache done by the victim. To perform the attack, the attacker first flushes the cache line it is targeting and waits for the victim to do an encryption. When the victim does the encryption, the T-table of AES may be cached in the target cache line. After the encryption, the attacker reloads the cache line and records the time of the reload. If the victim has accessed the cache line during the encryption, the time for the reload will be short, otherwise the victim has not accessed that cache line. If a hit has occurred, the attacker is able to load the victim's data. *Flush+Reload* has high accuracy because it target a specific cache line and a hit and miss can be easily determined. However, a high cache miss rate will make the encryption run slower and it could signal the victim that there is a side-channel attack.

2.3 Flush+Flush Attack

Another cache side-channel attack that has been proposed is the *Flush+Flush* attack, that also targets the LLC. This attack differs from *Flush+Reload* because it only depends on the time it takes to do a cache flush and it does not do any memory accesses. The attack can determine a cache hit or miss because the execution time of a flush on an empty cache is shorter than the execution time of a flush on a full cache. To perform the attack, the attack first flushes the cache, waits for the victim to do an encryption, then does a second flush of the cache. The time it takes to do both flushes are recorded. Again, if a hit has occurred, the attacker is able to load the victim's data. This attack is less detectable than the *Flush+Reload* attack because of the number of cache hits are reduced and there are no cache misses. Because of this, the victim will not be

aware of a side-channel. However, this attack is less accurate than the *Flush+Reload* attack because the timing difference between the hit and miss is lower and there is an increased risk of false positives.

3. Experiments

3.1. X86_64 Architecture

The COE Linux System, has a x86_64 architecture with and a shared last-level cache. The execution time of the flush and reload was determined using a cache profile benchmark. The `clflush` instruction is used to flush the cache block at the target address and the `maccess` instruction is used to access the target address. **Figure 1** shows the inline assembly `clflush` function and **Figure 2** shows the inline assembly `maccess` function.

Figure 1 - `clflush` Instruction

```
void clflush(volatile void* Tx) {
    asm volatile("lfence;clflush (%0) \n" :: "c" (Tx));
}
```

Figure 2 - `maccess` Instruction

```
static __inline__ void maccess(void *p) {
  asm volatile("movq (%0), %%rax\n" : : "c"(p) : "rax");
}
```

Flush+Reload

To implement the *Flush+Reload* attack, we have our program perform 1 million encryptions and collect the ciphers that produce a cache hit. The attacker and victim programs are compiled with the same shared library, `libcrypto.so`, provided by the OpenSSL library. Our `doTrace()` function first generates a plaintext, flushes the cache at our target address (the address of the shared library + the address of the first entry of the T-table used for the encryption), performs AES encryption (provided by the OpenSSL library), and times the reload of the target address after encryption. If the time of the reload is under a certain threshold, the ciphertext is saved for future analysis. **Figure 3** shows our `doTrace()` function.

Figure 3 - `doTrace()` Function for *Flush+Reload* attack

```
void doTrace()
{
      // generate a new plaintext
     generatePlaintext();

     clflush(target);
```

```
      // do encryption
      AES_KEY expanded;
      AES_set_encrypt_key(key, 128, &expanded);
      AES_encrypt(plaintext, ciphertext, &expanded);

       // record timing and ciphertext
      *timing = reload(target);
      saveTrace();

      // keep the ciphertext where its time is lower than threshold
      int i;
      int threshold = 150;
      if (*timing < threshold) {
            printText(ciphertext, 16, "ciphertext");
            printf("Timing: %i\n", *timing);
            fwrite(ciphertext, sizeof(uint8_t), 16, keptFP);
      }
}
```

Flush+Flush

To perform the *Flush+Flush* attack, the attacker first profiles the cache to estimate the cache him and miss threshold . In the `doTrace()` function, the cache at the target address is flushed, a plaintext is created then encrypted to produce the ciphertext. The cache is again flushed and this operation is timed. If the time is above the threshold, the ciphertext is saved in a separate text file. The target address here is the address of the shared library + address of Te0 table entry. The text file with the stored cipher text is now analysed to retrieve the key. **Figure 4** shows the doTrace function that performs the *Flush+Flush* attack.

Figure 4 - doTrace() Function for *Flush+Flush* attack

```
void doTrace()
{
      volatile uint32_t time;
       uint64_t t1, t2;

     // generate a new plaintext
     generatePlaintext();

     // set the cache to a known state
     clflush(target);

     // do encryption
      AES_KEY expanded;
```

```
     AES_set_encrypt_key(key, 128, &expanded);
     AES_encrypt(plaintext, ciphertext, &expanded);

     t1=timer();
     clflush(target);
    t2=timer();

    // record timing and ciphertext
    *timing = t2-t1;
    saveTrace();

     // keep the ciphertext where its time is lower than threshold
     int i;
     int threshold = 351;
     if (*timing < threshold) {
          printText(ciphertext, 16, "ciphertext");
          printf("Timing: %i\n", *timing);
          fwrite(ciphertext, sizeof(uint8_t), 16, keptFP);
     }
```

Key Recovery

To retrieve the encryption key, our `offline_analysis.c` program reads in the ciphertext values that produced cache hits. The program loops through each byte of the ciphertext and performs the following calculation:

$$s = s^{-1}[ki \text{ xor } cj] \qquad \textbf{(1)}$$

where k_i are key guesses from 0 to 255 and s^{-1} is the inverse of the sbox matrix from the AES encryption algorithm. If the *s* value is greater than 16 or less than 0, the program counts that this is a possible key byte. The key guess counts are recorded in a count buffer. After analysis, the top four key guess counts are determined to be four bytes of the encryption key. **Figure 5** shows the code that caclulates *s* and **Figure 6** shows the code that determines a key byte using the count buffer.

Figure 5 - Inverse sbox calculation

```
int cypher_byte;
for (i = 0; i < 16; i++) {
     cypher_byte = ciphertext[i];

     //loop through each key guess
     int k, s;
     for(k = 0; k < 256; k++) {
          s = rsbox[k ^ cypher_byte];
          if (s > 0 && s < 16)
```

```
                count[k]++;
     }
}
```

Figure 6 - Retrieve key bytes

```
int max = count[0];
int key_byte1;
for (i = 1; i < 256; i++) {
     if (count[i] > max) {
          max = count[i];
          key_byte1 = i;
     }
}
count[key_byte1] = 0;
printf("Key byte1: %d\n", key_byte1);
```

3.2. ARM Architecture

The *Flush+Reload* and *Flush+Flush* attacks are also demonstrated on the COE ARM server, which is 64-bit ARMv8 (AArch64). Both attacks require Flushing the cache, timing the operations, and accessing cache lines. These instructions are architecture specific and thus is implemented differently in ARM.

Flushing the cache

Flushing is achieved in ARM by making use of the system instructions `DC CIVAC`(DC - Data Cache, CIVAC - Clean and Invalidate by Virtual Address to point of Coherency).
The instruction `DSB ISH - DSB` (Data Synchronization Barrier) makes sure that the instructions following it will start executing only when all those instructions before `DSB` has completed. Specifying the option `ISH` (Inner Shareable domain) applies this barrier for instructions in this shareable domain. This is similar to the instruction `lfence` (Load fence) in x86. ISB - Instruction Synchronization barrier makes sure that all future instructions are fetched from memory or cache and not from pipeline. This is important as we need to time the access to the cache. So, ISB flushes the pipeline. Figure 7 shows the flush instruction done in ARM.

Figure 7: `flush` instruction in ARM

```
void flush(volatile void* Tx) {
     asm volatile ("DC CIVAC, %0" :: "r"(Tx));
     asm volatile ("DSB ISH");
     asm volatile ("ISB");
}
```

Timing the Reload

Unlike in x86, no assembly function is used to calculate the time in ARM. We use the system call `clock_gettime()` to obtain the time elapsed. The high precision clock, `CLOCK_MONOTONIC` is used as the clock id. **Figure 8** shows the reload function and the timing.

Figure 8 - Reload timing

```
uint32_t reload(void *target)
{
    uint64_t diff;
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start); /* mark start time */
    maccess(target);
    clock_gettime(CLOCK_MONOTONIC, &end); /* mark start time */
    diff = end.tv_nsec - start.tv_nsec;
    return(diff);
}
```

Accessing the cache line

The specified address is accessed by a simple pseudo instruction **`LDR`** that loads the register with the address value. This load instruction access the memory location specified and the time taken to access is analysed. **Figure 9** shows the maccess function used in ARM.

Figure 9 - Access the target Address

```
static __inline__ void maccess(void *p) {
     volatile uint32_t value;
     asm volatile ("LDR %0, [%1]\n\t"
      : "=r" (value)
      : "r" (p) );
}
```

Flush+Reload *and* Flush+Flush *Attacks*

The *Flush+Reload* and *Flush+Flush* attacks are implemented in the same way it is done in x86. See Section **3.1**.

Key Recovery

The same method to retrieve the key on the x86 architecture is also used to retrieve the key in this architecture. This is demonstrated in our `offline_analysis.c` program shown in **Figure 5** and **Figure 6** in section **3.1**.

4. Results

4.1 x86 profiling and results for both the attacks

Figure 10 below shows the cache profiling results for x86. For *Flush+Reload* we need to differentiate the time between cache hits and misses. The profiling results are plotted. From the graph the suitable threshold value is identified to be 150.

Figure 10: Cache hit and miss threshold graph for flush and reload

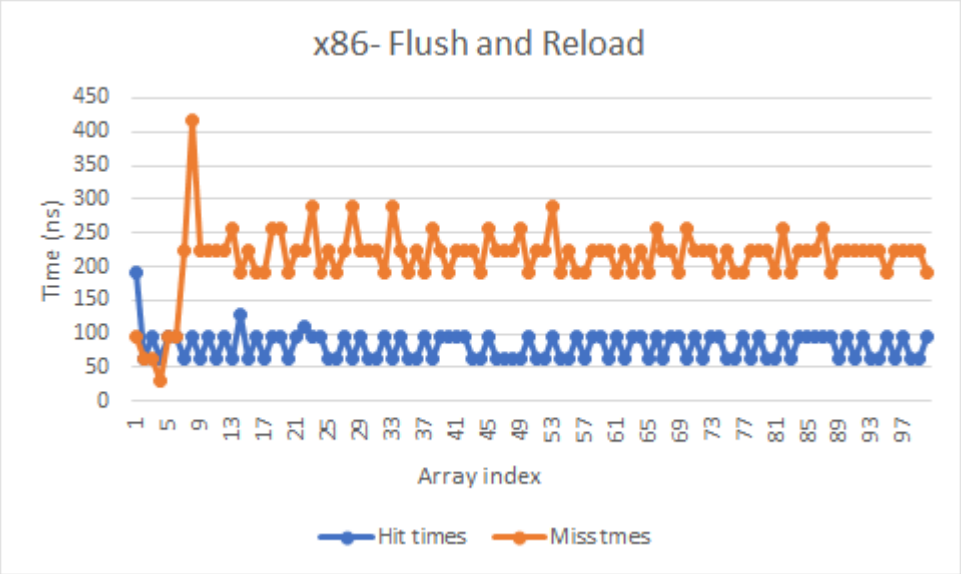


Figure 11 below shows the result of the *Flush+Reload* attack on x86. The recovered four key bytes and the saved ciphertext (those with hits) are printed in the output screen.

Figure 11: Recovered key and Ciphertext using *Flush+Reload* on x86

```
82 d6 65 e0 01 99 3b 19 a1 e8 dc 19 10 76 b2 e2
08 3c 06 5d 98 29 6a 13 f7 72 67 e9 53 81 bc 95
f4 45 8e 63 66 b6 de c0 84 a4 d3 a5 94 09 d5 b6
94 6d ff dc 40 33 c8 7a f6 4f 2f 2c f0 a4 6c 6f
94 d1 7a 31 ba 32 1c c2 2c f0 04 7e 1b 3a ac 96
8f 15 2c b7 c7 75 e8 69 39 28 c9 a5 94 2d 75 81
66 e2 d3 68 f4 73 a9 1d db 88 c4 9f 56 f9 b8 d3
Key byte1: 69
Key byte2: 73
Key byte3: 142
Key byte4: 245
nagulan@degrees:~/CA/project/fnf$
```

For *Flush+Flush*, the time difference between flushing a data-filled cache and a en empty cache is to be identified. The graph in **Figure 12** below is plotted to find this threshold. From the graph, the threshold for flush and flush in x86 was calculated to be 350.

Figure 12: Flush and Flush cache profile for x86

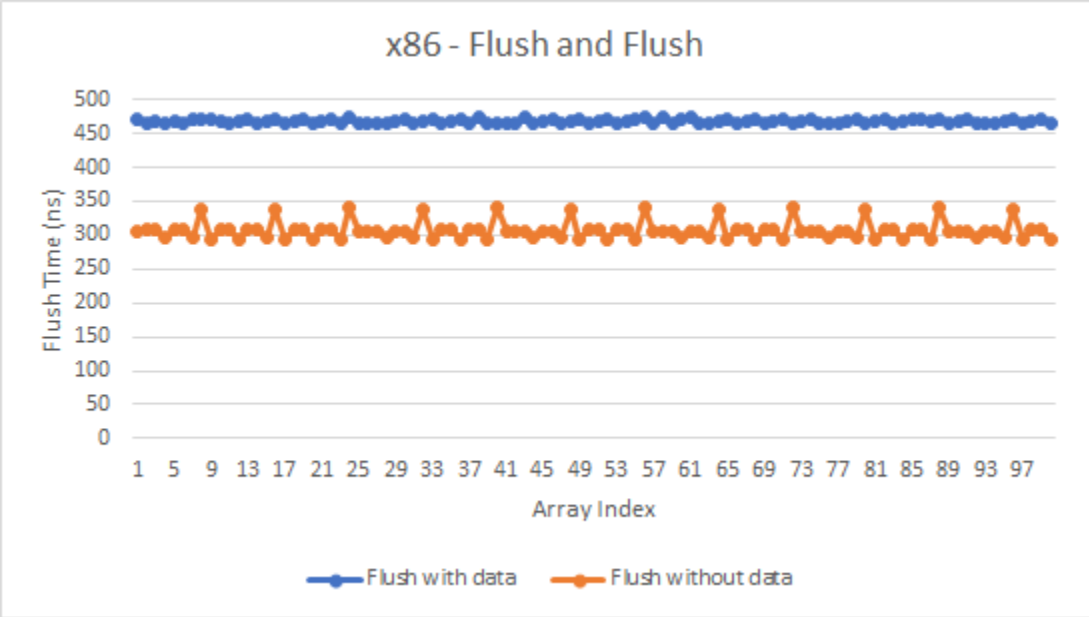


Figure 13 below is a sample screenshot of the output from the flush and flush attack. The output screen prints out the "ciphertexts" for which there were cache hits (in this case,longer flushing time). 4 key bytes were recovered using the `offline_analysis.c` program and are printed out in the result.

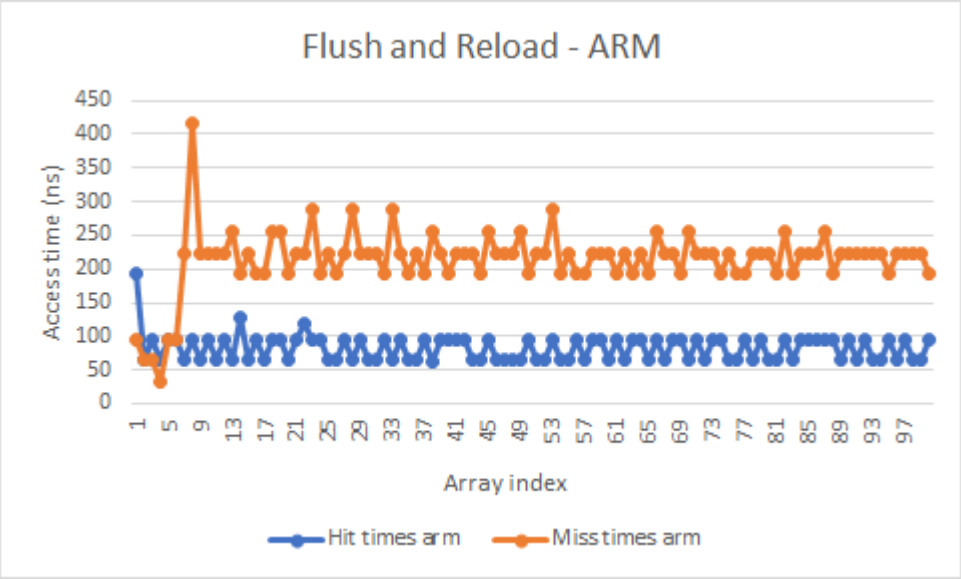
Figure 13: Recovered key using Flush and Flush attack on x86

```
a7 8b 5d 71 dc ea f4 2e cc 52 17 13 49 11 58 4b
16 75 cd d2 8e e6 78 38 5e 4c 89 28 41 8a 22 58
31 eb 00 1e 40 98 6c 47 0a f5 9e 54 a7 2b 0b 40
fd 67 d6 01 ab 98 43 ab 13 0b c1 80 69 ea c1 3b
Key byte1: 48
Key byte2: 204
Key byte3: 230
Key byte4: 7
```

4.2 ARM profiling and Results for both the attacks.

Just like in x86, the hits and miss times of the cache is measured in the ARM architecture. The values are plotted as shown in **Figure 14**.

Figure 14: Cache profiling on ARM for flush and reload



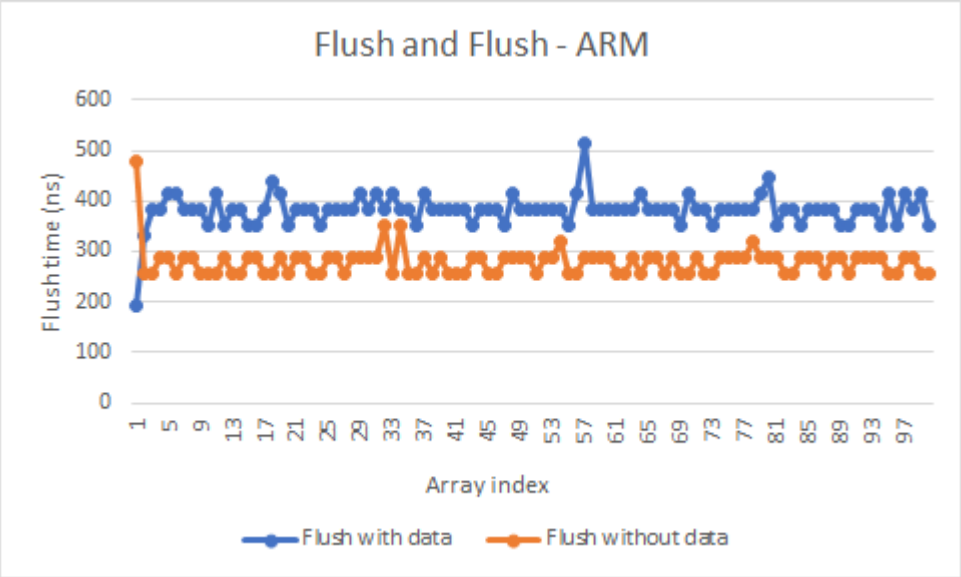
As it can be seen from the figure, the threshold for miss and hit time is identified as 150. The saved ciphertext (those with hits) are printed along with the recovered key bytes in the sample output screen shown in **Figure 15** below.

Figure 15: Flush and Reload recovered key bytes on ARM

```
33 fc f0 55 7d ed 2a 3f e7 20 67 49 2f 98 0a 81
cd 49 55 23 94 df ac 8f 8a de 4f c5 92 3e 6c cb
df be a3 0f 82 2b 3c 1e 2c bd 00 73 db d7 35 a5
97 c9 a1 c4 35 c7 83 76 be b2 d8 75 93 47 24 38
Key byte1: 30
Key byte2: 150
Key byte3: 66
Key byte4: 2
nagulan@degrees:~/CA/project/arm_attack$
```

Like the profiling in x86, the time difference between flushing a data-filled and empty cache in the ARM architecture is found through the graph in **Figure 16** below.

Figure 16: Flush and Flush cache profiling on ARM



The suitable threshold for *Flush+Flush* on ARM is identified as 310.

The saved ciphertext and the recovered key bytes are printed in the sample output in **Figure 17**.

Figure 17: Flush and Flush recovered keys on ARM

```
59 da 69 cf e5 2e 65 85 4d dc 15 75 d6 c9 21 95
6b 11 c5 dd 2d 64 32 e7 55 0a 4c 4d eb 03 8c 51
73 c0 6b 5d 47 7c a7 0c 78 ee 83 c1 48 27 1d 20
03 2c 51 82 ee e3 06 2c fc 2d a2 e5 53 f9 69 1c
Key byte1: 149
Key byte2: 147
Key byte3: 159
Key byte4: 197
nagulan@degrees:~/CA/project/arm_attack$
```

There are a total of four attacks demonstrated - *Flush+Reload* and *Flush+Flush* on two different architectures (x86 and ARM) and the recovered key bytes on all the attacks are tabulated in **Figure 18.**

Figure 18: Results Tabulation

Results	**x86**		**ARM**	
	Flush+Reload	Flush+Flush	Flush+Reload	Flush+Flush

Threshold	150	350	150	310
Recovered Key bytes	Byte1: 69 Byte2: 73 Byte3: 142 Byte4: 245	Byte1: 48 Byte2: 204 Byte3: 230 Byte4: 7	Byte1: 30 Byte2: 150 Byte3: 66 Byte4: 2	Byte1: 149 Byte2: 147 Byte3: 159 Byte4: 197

5. Conclusion

This paper demonstrates cache side-channel attacks against AES encryption on two different architectures, x86 and ARM. There are two different side-channel attacks used, *Flush+Reload* and *Flush+Flush* attacks. The experiments carried out profiles to identify the hit times, miss times and time taken to flush the cache. These thresholds are used to perform the attack. The four bytes of the encryption key are recovered from each of the attack using results from a million encryptions. All the experiments were performed on the x86 and ARM coe system.

Our experiments did not verify the accuracy of the recovered key, due to limited time and resources. In order to verify the key, the complete 16 byte key has to be recovered, which requires more samples. To get the plaintext, an AES decryption must be done using the recovered key and ciphertext. If the plaintext matches the original plaintext, we can confirm that the recovered key is correct. We were only able to retrieve four key bytes with confidence in our trace.

6. References

[1] YaromY, Falkner K. Flush+ reload: a high resolution, low noise, L3 cache side-channel attack. USENIX Security 2014
[2] GrussD, Maurice C, Wagner K. Flush+ Flush: A Stealthier Last-Level Cache Attack. arXiv preprint arXiv:2015
[3] J. Bonneau and I. Mironov: Cache-Collision Timing Attacks against AES, Workshop on Cryptographic Hardware and Embedded Systems — CHES 2006, LNCS, Springer Verlag
[4] "Advanced Encryption Standard", En.wikipedia.org, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard. [Accessed: 06- Dec- 2019].
[5] "ARM Information Center", Infocenter.arm.com, 2019. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/Babbfdih.html. [Accessed: 06- Dec- 2019].
[6]"ARM Information Center", Infocenter.arm.com, 2019. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/CIHGHHIE.html. [Accessed: 06- Dec- 2019].
[7]I. OpenSSL Foundation, "/index.html", Openssl.org, 2019. [Online]. Available: https://www.openssl.org/. [Accessed: 06- Dec- 2019].

[8]"openssl/openssl", *GitHub*, 2019. [Online]. Available: https://github.com/openssl/openssl.git. [Accessed: 06- Dec- 2019].

7. Project Proposal

1. We want to study side channel attacks on the cache memory in different cache architectures.
2. We plan to use the College of Engineering linux server to demonstrate the attacks. We will also explore doing side channel attack on mobile phones with the ARM architecture and on our personal windows machines. The Armageddon, OpenSSL library and AES encryption will be used to implement the attacks.
3. We will run two different types of side channel attacks, “Flush+Reload” and “Flush+Flush”, on ARM and x86. This will show the vulnerability of different cache architectures. We also intend to find the time taken to retrieve the encryption key.
4. The results we will retrieve are the timing distributions of cache hits and misses (memory access time), the encryption key, and the accuracy of the encryption key. Other metrics will be determined when we start the project.

   A = Demonstrate both “Flush+Reload” and “Flush+Flush” attack on both architectures (ARM and x86), necessary results produced and compared in the project report.
   A- = Demonstrate both “Flush+Reload” and “Flush+Flush” attack on x86 and one of the attacks on ARM. Compare results and record them in the report.
   B+ = Demonstrate both “Flush+Reload” and “Flush+Flush” attack on one system and report the results in the project report.
   B = Demonstrate one attack on one system and report the results on the project report.