Modified Cache Template Attack on AES

Mahdi Esfahani[1], Hadi Soleimany[2], and Mohammad Reza Aref[3]

[1] Department of Mathematics, Karaj Branch, Islamic Azad University, Karaj, Iran
[2] Cyberspace Research Institute, Shahid Beheshti University, Iran
Email: h_soleimany@sbu.ac.ir
[3] Department of Electrical Engineering, Sharif University of Technology, Tehran , Iran

Abstract. CPU caches are a powerful source of information leakage. To develop practical cache-based attacks, there is an increasingly need to automate the process of finding exploitable cache-based side-channels in computer systems. Cache template attack is a generic technique that utilizes Flush+Reload attack in order to automatically exploit cache vulnerability of Intel platforms. Cache template attack on T-table-based AES implementation consists of two phases including the profiling phase and the key exploitation phase. Profiling is a preprocessing phase to monitor dependencies between the secret key and behavior of the cache memory. In addition, the addresses of T-tables can be obtained automatically. In the key exploitation phase, most significant bits (MSBs) of the secret key bytes are retrieved by monitoring exploitable addresses. In this paper, we propose a simple yet effective searching technique which accelerates the profiling phase by a factor of at most 64. To verify the theoretical model of our technique, we implement the described attack on AES. The experimental results showed the profiling phase runtime of the cache template attack is around 10 minutes while our method speeds up the running of this phase to around 9 seconds.

Keywords: cache memory, Flush+Reload attack, inclusive memory, AES, cache template attacks, automation

1 Introduction

Cryptographic algorithms and protocols alone do not provide data security, but they require a digital platform to run securely and efficiently. Evaluation of the security of cryptographic algorithms against side-channel attacks is one of the most important challenges in the field of applied cryptography. Unlike the mathematical analyses which use structural weaknesses in cryptographic primitives, side-channel attacks exploit the data leaking from the implementation of cryptographic algorithms.

Timing variations during the run-time program is one of the most important sources for the leakage in the timing channels. Memory accesses and the presence of branch in programs are costly at the runtime. Thus, modern processors use cache memories and branch predictor to reduce this cost. Such an optimization at runtime leads to timing variations. Easy measurement and the lack of need for specific hardware tool for measuring are among the specific features of timing side-channel attacks. The cache-based side channel attacks require distinguishing between cache hit and cache miss events by measuring the execution time of the target cryptographic algorithm. The execution time difference between a cache hit and cache miss leads to information leakage. Cache-based side-channel attacks are classified into three categories of time-driven, trace-driven, and access-driven attacks. In time-driven attacks [1, 2], the attacker does not have access to the cache and he knows the capacity of the cache memory lines. Also, the attacker should retrieve the secret key by only measuring the cryptosystem runtime. In access-driven attacks, the attacker is able to evict or reload data from the cache memory [3, 4, 5]. Access-driven attacks are classified into synchronous and asynchronous categories. In synchronous attacks, the attacker is able to trigger encryption or decryption but in asynchronous attacks, the activities of the non-privileged adversary are performed in parallel to the victim [6, 7, 8]. In trace-driven attacks, the attacker observes a series of cache misses and cache hits during encryption [9].

The first covert channel based on cache memory was proposed by Hu [10]. Kelsey believed attacks based on cache hit ratio in ciphers with large S-boxes are possible [11]. The cache-based side-channel attacks on the implementations of the ciphers with large lookups were considered by Tsunoo and the first results of experimental attacks on the block ciphers such as DES were obtained [12]. The aggregate number of cache hits and misses through indirect measurements of the total execution time of the encryption process was used by Bernstein in order to attack AES for the first time [13]. After that, several practical time-driven cache attacks on AES have been proposed [14, 15]. In the access-driven attacks, Percival et al. were the pioneers in the access-driven attacks on RSA and AES [16]. Yarom and Falkner [17] proposed the Flush+Reload attack and successfully applied the attack on the implementation of RSA. Ronen, Gillham, Genkin and et al. [18, 19] performed the Flush+Reload attack on the targets via the last level cache in the virtualized environments. After that, the Flush+Reload technique has been applied on AES in order to retrieve all 16 bytes of AES in the native and cross-VM environments, respectively [20, 21]. In the next year, Golmezoglu, Inci, Irazoqui and et al. [22] improved the attack [21] by guessing the possible candidates for the last round of AES which leads decreasing the noise of the attack.

In most of the proposed attacks, the attacker needs to identify the vulnerabilities manually which can be a notable limitation. In response to this challenge, Gross et al. proposed cache template attacks [23]. The attack makes use of the Flush+Reload technique in order to automatically exploit cache-based vulnerabilities in a program running on architecture with shared inclusive last-level caches.

1.1 Our Contribution

The cache-template attack on the T-table based implementation of AES proposed in [23] performs both profiling and exploitation phases automatically. The high runtime of the profiling phase is an important limiting factor in the proposed attack. Measuring the cache-hit ratio is the most expensive step in the attack.

To increase the runtime speed of the profiling phase, we introduce a simple yet efficient method in this paper. More precisely, we propose a more efficient method to measure the cache-hit ratio for each address of the attacked binary and construct the profile.

Outline. This paper is organized as follows. In Section 2, we provide background information. We describe the cache template attacks in Section 3. In Section 4, we give an overview of our technique and also discuss the experimental results. Finally, we conclude in Section 5.

2 Background

2.1 CPU Caches

Cache is an essential feature of modern architecture and increases the speed of memory access by keeping recently-accessed instructions and data. The cache memory is organized as multiple cache sets, each consisting of a fixed number of cache lines [24]. Each cache line is split into a *tag*, *index* and *block offset*. The index is used to map specific memory locations to the sets of the cache memory. The most significant bits of the address determine the tag which is used to uniquely identify a specific cache line in a cache set. The block offset identifies a particular location within a cache line.

In order to bridge the gap between data retrieval speed and processor speed, modern processors exploit a hierarchy in the structure of the cache. Closest to the core is the L1 cache which consists of separate parts for data and instructions while other levels are unified. Down to the last-level cache (LLC), the cache level gets larger and slower. the Last-level cache is generally shared between the cores. In most of the Intel processors, the cache memory has three levels and LLC is inclusive which means all data in the L1 and L2 caches are also present within the L3 cache [25]. L3 cache is shared among all cores and the inclusive cache can be utilized to apply Flush+Reload attack [22, 21, 26, 27] which will be described in the next section.

2.2 Flush+Reload Attack

In Flush+Reload attack on the Intel system, an attacker flushes cache memory using the `clflush` instruction. The Flush+Reload attack on cryptographic algorithms makes use of the shared memory and libraries features in the L3 cache between the attacker and the victim program. Flush+Reload works as follows:

1. The attacker maps shared library (or binary) into virtual address space and accesses shared library for loading into the cache.

2. The attacker flushes shared library from the cache and waits for sufficient time for the victim to use (or not use) the memory locations that he has flushed.
3. After the victim has been scheduled, the attacker reloads the previously flushed shared library and measuring the load time.

The main observation is that if the victim did not access to the data flushed in the second step, the data is not available in the cache memory and consequently the attacker measures high latency. There are two main reasons regarding that the Flush + Reload method is more powerful than the previous access-driven attacks. First, unlike previous attacks that based on the cache set, the attacker has access to the cache line in the Flush+Reload attack which leads to increase the accuracy. Second, the Flush + Reload attack is a cross-core attack as the L3 cache is shared between all processor cores. For this reason, in recent years, the Flush+Reload method has been used in many cache-based attacks [28, 29, 30, 31, 32].

2.3 Memory Access in AES Implementations

Advanced Encryption Standard (AES) has been adopted as an encryption standard by the U.S. government [33]. AES has a substitution-permutation network (SPN) structure with a fixed block size of 128 bits, and a key size of 128, 192 or 256 bits. In this paper, we consider an attack on AES-128. AES operates on a 4×4 order array of bytes which termed the state matrix and most calculations are done in $GF(2^8)$. AES-128 has ten rounds. Each round composed of four types of transformation namely, SubByte, ShiftRows, MixColumns and AddRoundKey. Exceptionally, the last round does not have MixColumns.

In order to increase the speed and efficiency of software implementation, different methods have been used in hardware and software. As the SubBytes is the most expensive to implement, in the software implementation, the look-up table is ideal to implement this operation. However, a well-known method T-table implementation [33] has been adopted by several crypto-libraries such as OpenSSL which precomputes the round function. In the T-table implementation, four look-up tables are considered as follow:

$$T_0(z) = \begin{bmatrix} 02.S(z) \\ S(z) \\ S(z) \\ 03.S(z) \end{bmatrix}, T_1(z) = \begin{bmatrix} 03.S(z) \\ 02.S(z) \\ S(z) \\ S(z) \end{bmatrix}, T_2(z) = \begin{bmatrix} S(z) \\ 03.S(z) \\ 02.S(z) \\ S(z) \end{bmatrix}, T_3(z) = \begin{bmatrix} S(z) \\ S(z) \\ 03.S(z) \\ 03.S(z) \end{bmatrix} \quad (1)$$

Each table maps a byte z to a 32-bit value. Consequently, the size of the each T-table is 1024 bytes. If each cache line size is 64 bytes, 16-cache lines are required in order to store one T-table. Using the T-tables presented in Equation 1, one can express the the first nine rounds of AES as it is described in Equation 2.

$$T_0[s_0^{(r)}] \oplus T_1[s_5^{(r)}] \oplus T_2[s_{10}^{(r)}] \oplus T_3[s_{15}^{(r)}] \oplus [k_0^{(r)} k_1^{(r)} k_2^{(r)} k_3^{(r)}] \|$$

$$T_0[s_4^{(r)}] \oplus T_1[s_9^{(r)}] \oplus T_2[s_{14}^{(r)}] \oplus T_3[s_3^{(r)}] \oplus [k_4^{(r)} k_5^{(r)} k_6^{(r)} k_7^{(r)}] \|$$
$$T_0[s_8^{(r)}] \oplus T_1[s_{13}^{(r)}] \oplus T_2[s_2^{(r)}] \oplus T_3[s_7^{(r)}] \oplus [k_8^{(r)} k_9^{(r)} k_{10}^{(r)} k_{11}^{(r)}] \|$$
$$T_0[s_{12}^{(r)}] \oplus T_1[s_1^{(r)}] \oplus T_2[s_6^{(r)}] \oplus T_3[s_{11}^{(r)}] \oplus [k_{12}^{(r)} k_{13}^{(r)} k_{14}^{(r)} k_{15}^{(r)}] \quad (2)$$

where $s_i^{(r)}$ represents the i-th byte of the state in the $r-$th round in which $0 \leq r \leq 9, 0 \leq i \leq 15$. The final round cannot use the tables presented in Equation 2 due to the absence of the MixColumns operation [34]. In standard implementations, there are two strategies for implementing the final round. One method is defining another table for the last round. Another method is using the tables presented in Equation 1 partially. Since the final round comprises only two operations on the state: SubBytes and ShiftRows, the values $S(s_{12}^9)$, $S(s_8^9)$, $S(s_4^9)$, and $S(s_0^9)$ can be calculated by accesses to table T_0 and using the second element of T_0. Similarly, other bytes can be calculated by accesses to tables T_1, T_2 and T_3. The implementation of the final round in the OpenSSl library version 1.1.0f uses the second method.

3 Cache Template Attacks

Cache template attack on the first round of the AES cipher was proposed in [23]. The attack consists of two phases: 1) profiling phase, and 2) key exploitation phase. In the profiling phase, dependencies between the processing of the secret key of the AES and specific cache accesses are determined. Also, in this phase the attacker can accurately determine the start and end of the T-table AES in `libcrypto.so` file of the OpenSSL through the cache hit ratio. In the key exploitation phase, the most significant bits (MSBs) of the key of each byte is retrieved. In what follows we briefly describe both phases.

3.1 Profiling Phase

The profiling phase measures the cache-hit ratios on specific addresses during the execution of the AES. The cache-hit ratios are stored in a matrix which has one column per encryption and one row per address. The matrix is namely called *cache template matrix*. In order to compute cache template matrix, first, AES encryption is performed to encrypt a plaintext in which a specific byte is a constant and fixed value while other bytes can be arbitrary bytes randomly.

If each cache line size is 64 byte, then for each key byte k_i, the upper 4 bits of k_i can be profiled. We need to profile 16 addresses for each key byte k_i. For example, to determine the cache template matrix for each value of the key byte k_0, the attacker flush the content of the first address in the binary file and perform AES with a fixed key k_0. The first byte of plaintext p_0 is chosen $0x00$ and other bytes of plaintext, i.e $(p_1, \cdots, p_{15})$ are randomly chosen. Then, the content of the same address is accessed and the execution time is measured. If the access time is less than the threshold can be interpreted as cache hit with high probability. The attacker performs several times the process and compute the cache-hit ratio on the same address. During the attack process, the cache-hit ratio for each address of binary file `libcrypto.so` is computed. Also, the attack process is repeated for different values $\{0x10, 0x20, ..., 0xF0\}$ of the first byte of plaintext p_0.

Each column vector of the cache template matrix is called a *profile*. We denote the j-th column vector of the matrix by $\overrightarrow{p_j}$ which is a profile for a constant value $p_0 = 16 \cdot j$ where $0 \leq j \leq 15$. Each row represents the address range of the T-table. In other words, each matrix element represents the cache hit ratio for a constant p_0 and a T-table address.

We should remove all rows which contain redundant information from the matrix. This can be done by pruning the rows which have a small difference between the minimum and maximum cache-hit ratio. One should monitor all addresses $a_{64 \cdot i}$ in the binary file `libcrypto.so`, during the execution of the AES in order to find the start and end of the T-tables in `libcrypto.so` file and create cache template matrix. For this reason, the runtime speed of the profiling phase is slow.

3.2 Exploitation Phase

The attacker performs encryption several times for different chosen plaintexts under an unknown key. 16-byte keys k_i are attacked sequentially where $0 \leq i \leq 15$. For example to retrieve the upper 4 bits of k_0, the plaintexts are chosen randomly, except the 4 upper bits of p_0 which are fixed to the same chosen value used in the profiling phase.

For all addresses in the cache template matrix resulting from the profiling phase, the cache activity is constantly monitored and cache hit ratio is stored in a vector $\overrightarrow{h}$. The attacker computes the similarity between $\overrightarrow{h}$ and each profile $\overrightarrow{p_j}$ from the cache template matrix using the mean square error function $S(\overrightarrow{h}, \overrightarrow{p}_j)$. Let us assume that for a profile j', $S(\overrightarrow{h}, \overrightarrow{p}_{j'})$ has the minimum value. Then we can conclude that for the plaintexts with the fixed value $p_0 \in \{0x00, 0x10, 0x20, ..., 0xF0\}$, the corresponding address of T-table is accessed. The address of the T-table that is accessed and corresponding to p_0 is determined in the profiling phase. By considering the profile $\overrightarrow{p}_{j'}$ we can determine which cache line has most cache-hit ratio and consequently the four most significant bits of $<s_0>$ can be computed. Finally, the four most significant bits of k_0 is exploited using Equation 3.

$$<k_0> = <p_0 \oplus s_0> \tag{3}$$

The four most significant bits for other bytes of the secret key can similarly be retrieved by utilizing the aforementioned method.

4 Our Attack Scenario

4.1 Modified Profiling Phase

In this section we present an effective method for monitoring the addresses of binary file `libcrypto.so` which accelerates the profiling phase in the cache template attack.

As it is described in Section 3.1, the distance between two addresses in the monitoring step of profiling phase is considered as 64 bytes by the authors of [23]. In the proposed approach, the attacker should consider all addresses in the binary file with the distance of 64 bytes and repeat a described process for each of them [23]. This approach requires a notable time. Increasing the distance between the addresses during the search step can be an appropriate solution but it is challenging as the address line of the first block of T_0 cannot be found in this way. On the one hand, restricting the distance between the addressed to 64 bytes slows down the speed. On the other hand, increasing the distance leads to missing the start point as 4096 bytes are allocated for saving the lookups. In order to overcome this challenge, we consider a combined approach.

Our approach consists of two steps. First, we trace addresses in which the difference between two consecutive addresses is d bytes where $d < 4096$ and stop the process when we could find an address in which the cache hit ratio is large enough. In the second step, we trace the addresses in the backward direction such that the distance between two consecutive addresses is 64 bytes and stop the process when the cache hit ratio is small. The benefit of this approach is obvious, as the process of finding the address of the first block of T_0 can be accelerated notably.

We assume a cache line size of 64 bytes. The process of modified profiling phase for the upper 4 bits of k_0 is described in Algorithm 1.

First we choose the byte p_0 from the set $\{0x00, 0x10, 0x20, ..., 0xF0\}$ and generate other bytes of plaintext randomly. Then we consider addresses $a_i = d \times i$ for $0 \leq i \leq e/d$ where e denotes the address of the last byte in the binary file B. We repeat the Flush+Reload process N times for each address a_i and the cache-hit ratio for a_i is computed and saved as $L[p_0][a_i]$. The *threshold* used in Algorithm 1 denotes the minimum cache miss cycles which depends on the processor. We stop the process if the cache-hit ratio for a specific $a_{start} = a_i$ is larger than $N/2$.

In the second step, we consider the addresses $\hat{a}_i = (a_{start} - 1024)64 \times i$. For each address, we repeat the Flush+Reload process N times and compute the cache-hit ratio $L[p_0][\hat{a}_i]$ and construct the cache template matrix $M[p_0][\hat{a}_i]$. We stop the process when the cache-hit ration for $\hat{a}_i$ is less than $N/2$.

4.2 Experimental Results

The benefit of our method and cache template attack, compared to existing cache attacks against AES [22, 21], is that they are fully automated. In addition, the cache template attack requires extremely low data (only 16–160 encryptions).

Algorithm 1 Profiling Phase Algorithm for k_0

```
Input: Binary file $B$, distance $d$ where $d < 4096$
Output: Cache template matrix $M$ for $k_0$
Map libcrypto.so file into memory
foreach $p_0 \in \{0x00, 0x10, 0x20, ..., 0xF0\}$ do
    for $(i = 0, i++, i \leq e/d)$ do
        $a_i = d \times i$
        for $(j = 1, j++, j < N)$ do
            $p_1 \parallel p_2 \parallel ... \parallel p_{15}$=rand()
            P $\longleftarrow (p_0 \parallel p_1 \parallel p_2 \parallel ... \parallel p_{15})$
            Flush $(a_i)$
            $AES_k(P)$
            $t_1 \longleftarrow$ start time
            Reload $(a_i)$
            $t_2 \longleftarrow$ end time
            $t_{p_0} = t_{p_0} + t_2 - t_1$
            $c_{p_0} = c_{p_0} + 1$
            $d_{p_0} = t_{p_0} / c_{p_0}$
            if $(d_{p_0} <$ threshold$)$ then
                $L[p_0][a_i]++$
            end
        end
    end
    if $L[p_0][a_i] \geq N/2$ then
        $a_{start} = a_i$ break
    end
end
foreach $(p_0 \in \{0x00, 0x10, 0x20, ..., 0xF0\})$ do
    for $(i = 0, i++, i \leq e)$ do
        $\hat{a}_i = (a_{start} - d) + 64 \times i$
        for $(j = 1, j++, j \leq N)$ do
            $p_1 \parallel p_2 \parallel ... \parallel p_{15}$=rand()
            P $\longleftarrow (p_0 \parallel p_1 \parallel p_2 \parallel ... \parallel p_{15})$
            Flush $(\hat{a}_i)$
            $AES_k(P)$
            $t_1 \longleftarrow$ start time
            Reload $(a_i)$
            $t_2 \longleftarrow$ end time
            $t_{p_0} = t_{p_0} + t_2 - t_1$
            $c_{p_0} = c_{p_0} + 1$
            $d_{p_0} = t_{p_0} / c_{p_0}$
            if $d_{p_0} >$ threshold then
                $L[p_0][\hat{a}_i]++$
            end
        end
    end
    if $L[p_0][\hat{a}_i] < N/2$ then
        break
    end
    $M[p_0][\hat{a}_i] = L[p_0][\hat{a}_i]$
end
```

Once the binary is deployed on the target system, it performs both profiling and exploitation phase automatically and returns the key byte candidates to the attacker. Also in contrast with the attacks [22, 21], in profiling phase of the cache template attack, the attacker does not need prior knowledge about the addresses of the T-table elements and he can accurately determine the start and end of the T-table implementation in the binary file.

In the profiling phase proposed in the original paper, to determine the cache template matrix M one key byte, the attacker should construct 16 profiles. To construct each profile, Flush+Reload technique is performed for the addresses a_i where $0 \leq i \leq \lceil \frac{e+1}{64} \rceil$ and each time repeated under N encryptions. Consequently, the the time complexity of the original method is $16 \times \lceil \frac{e+1}{64} \rceil \times N$ where e denotes the address of the last byte in the binary file and N denotes the number of required encryptions. In our approach, the time complexity of the profiling phase is approximately $16 \cdot \frac{e+1}{d} \times N + 16 \times N$ where d is the distance between two consecutive addresses. The time complexity is dominated by the term $16 \cdot \frac{e+1}{d} \times N$. Consequently, we expect our approach performs faster than original proposal by a factor of $\frac{d}{64}$ where $d < 4096$.

To verify the theoretical model, we implemented our approach and the original method which was proposed in [23]. The experimental results are summarized in Table 1. We tested our approach on the openSSL library (version 1.1.0f) under the Ubuntu 16:04 operating system performed in the Intel corei5-2.50 GHz.

In our experience, we consider different values for the distance between two consecutive addresses which is denoted by d. As it can be seen from Table 1, the profiling phase runtime of the cache template attack is around 10 minutes while our method speeds up the running of this phase to around 39, 23, 13 and 9 seconds for $d = 1024, 2048, 3072$ and 4096 bytes, respectively. Therefore, our approach is approximately 16, 32, 48 and 64 times faster than the original proposal for $d = 1024, 2048, 3072$ and 4096 bytes, respectively.

Table 1. Comparison results for the runtime speed (second) of the attacks

Number of the encryption	attack[23]	our attack ($d = 1024$)	our attack ($d = 2048$)	our attack ($d = 3072$)	our attack ($d = 4096$)
10	9.021 s	0.560	0.275	0.189	0.145
50	33.773 s	2.128	1.062	0.711	0.527
100	72.927 s	4.362	2.028	1.354	1.017
500	364.830 s = 6min	19.650	9.838	6.778	4.924
1000	681.916 s = 10min	39.343	23.090	13.216	9.832

5 Conclusion

Cache template attack is a method to automate the process of finding exploitable cache vulnerabilities. In this paper, we revisited the cache template attack on

T-table based implementation of AES and propose an efficient technique to speed up the process of profiling phase. Finally, we implemented our approach to experimentally validate the theoretical model. The experimental results confirm that our approach is faster than the original method.

Acknowledgments. This work was supported in part by the Iran National Science Foundation under Grant 96-53979 and in part by the Iran Telecommunication Research Center.

Bibliography

[1] Buch, D.H. and Bhatt, H.S. "Trinetra: a solution to handle cross-VM time-driven attack", *SN Applied Sciences.*, **2**(4), pp. 1–12 (2020).

[2] Yarom, Y. Genkin, D. and Heninger, N. "CacheBleed: a timing attack on OpenSSL constant-time RSA", *Journal of Cryptographic Engineering.*, **7**(2), pp. 99–112 (2017).

[3] Götzfried, J. Eckert, M. Schinzel, S. and et al. "Cache attacks on Intel SGX", *Proceedings of the 10th European Workshop on Systems Security.*, pp. 1–6 (2017).

[4] Xinliang, M. Liehui, J. and Rui, C. "Survey of Access-Driven Cache-Based Side Channel Attack", *Journal of Computer Research and Development.*, **57**(4), pp. 824 (2020).

[5] Schwarz, M. "Software-based Side-Channel Attacks and Defenses in Restricted Environments", (2019).

[6] Chen, S. Liu, F. Rui, C. and et al. "Leveraging hardware transactional memory for cache side-channel defenses", *Proceedings of the 2018 on Asia Conference on Computer and Communications Security.*, pp. 601–608 (2018).

[7] Chattopadhyay, S. Beck, M. Rezine, A. and et al. "Quantifying the information leakage in cache attacks via symbolic execution", *ACM Transactions on Embedded Computing Systems (TECS).*, **18**(1), pp. 1–27 (2019).

[8] Lapid, B. and Wool, A. "Navigating the Samsung TrustZone and Cache-Attacks on the Keymaster Trustlet", *European Symposium on Research in Computer Security.*, pp. 175–196 (2018).

[9] Ge, Q. Yarom, Y. Cock, D. and et al. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware", *Journal of Cryptographic Engineering.*, **8**(1), pp. 1–27 (2018).

[10] Hu, W.M. "Lattice scheduling and covert channels", *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy.*, pp. 52–61 (1992).

[11] Kelsey, J. Schneier, B. Wagner, D. and et al. "Side channel cryptanalysis of product ciphers", *European Symposium on Research in Computer Security.*, pp. 97–110 (1998).

[12] Tsunoo, Y. Saito, T. Suzaki, T. and et al. "Cryptanalysis of DES implemented on computers with cache", *International Workshop on Cryptographic Hardware and Embedded Systems.*, pp. 62–76 (2003).

[13] Bernstein, D.J. "Cache-timing attacks on AES", (2005).

[14] Acıiçmez, O. Schindler, W. and Koç, Ç.K "Cache based remote timing attack on the AES", *Cryptographers' track at the RSA conference.*, pp. 271–286 (2007).

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

[16] Percival, C. "Cache missing for fun and profit", *BSDCan.*, (2006).

[17] Yarom, Y. and Falkner K. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack", *23rd {USENIX} Security Symposium ({USENIX} Security 14).*, pp. 719–732 (2014).

[18] Ronen, E. Paterson, k.G. and Shamir, A. "Pseudo constant time implementations of TLS are only pseudo secure", *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.*, pp. 1397–1414 (2018).

[19] Ronen, E. Gillham, R. Genkin, D. and et al. "The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations", *2019 IEEE Symposium on Security and Privacy (SP).*, pp. 435–452 (2019).

[20] Inci, M.S. Gulmezoglu, B. Irazoqui, G. and et al. "Cache attacks enable bulk key recovery on the cloud", *International Conference on Cryptographic Hardware and Embedded Systems.*, pp. 368–388 (2016).

[21] Irazoqui, G. Inci, M.S. Eisenbarth, T. and et al. "Wait a minute! A fast, Cross-VM attack on AES", *International Workshop on Recent Advances in Intrusion Detection.*, pp. 299–319 (2014).

[22] Gülmezoğlu, B. Inci, M.S. Irazoqui, G. and et al. "A faster and more realistic flush+ reload attack on AES", *International Workshop on Constructive Side-Channel Analysis and Secure Design.*, pp. 111–126 (2015).

[23] Gruss, D. Spreitzer, R. and Mangard, S. "Cache template attacks: Automating attacks on inclusive last-level caches", *24th {USENIX} Security Symposium ({USENIX} Security 15).*, pp. 897–912 (2015).

[24] Ge, Q. Yarom, Y. Li, F. and et al. "Contemporary processors are leaky and there is nothing you can do about it", *The Computing Research Repository. arXiv.*, (2016).

[25] Gruss, D. Maurice, C. and Mangard, S. "Rowhammer. js: A remote software-induced fault attack in javascript", *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.*, pp. 300–321 (2016).

[26] Irazoqui, G. and Guo, X. "Cache Side Channel Attack: Exploitability and Countermeasures", *Black Hat Asia.*, **2017**(3), (2017).

[27] Saileshwar, G. and Qureshi, M.K. "Lookout for Zombies: Mitigating Flush+ Reload Attack on Shared Caches by Monitoring Invalidated Lines", *arXiv preprint arXiv:1906.02362.*, (2017).

[28] Lipp, M. Schwarz, M. Gruss, D. and et al. "Meltdown", *arXiv preprint arXiv:1801.01207.*, (2018).

[29] Schwarz, M. Schwarzl, M. Lipp, M. and et al. "Netspectre: Read arbitrary memory over network", *European Symposium on Research in Computer Security.*, pp. 279–299 (2019).

[30] Kocher, P. Horn, J. Fogh, A. and et al. "Spectre attacks: Exploiting speculative execution", *2019 IEEE Symposium on Security and Privacy (SP).*, pp. 1–19 (2019).

[31] Minkin, M. Moghimi, D. Lipp, M. and et al. "Fallout: Reading kernel writes from user space", *arXiv preprint arXiv:1905.12701.*, (2019).

[32] Seddigh, M. and Soleimany, H. "Enhanced Flush+ Reload Attack on AES", *ISeCure.*, **12**(2), (2020).

[33] Daemen, J. and Rijmen, V. "The design of Rijndael: AES-the advanced encryption standard", *Springer Science & Business Media.*, (2013).

[34] Rebeiro, C. Mukhopadhyay, D. and Bhattacharya, S. "Timing channels in cryptography: a micro-architectural perspective", *Springer.*, (2014).