Journal of Hardware and Systems Security (2018) 2:33–50
https://doi.org/10.1007/s41635-017-0025-y

A Survey of Side-Channel Attacks on Caches and Countermeasures

Yangdi Lyu[1] · **Prabhat Mishra**[1]

Received: 29 August 2017 / Accepted: 9 November 2017 / Published online: 27 November 2017

Abstract
With the increasing proliferation of Internet-of-Things (IoT) in our daily lives, security and trustworthiness are key considerations in designing computing devices. A vast majority of IoT devices use shared caches for improved performance. Unfortunately, the data sharing introduces the vulnerability in these systems. Side-channel attacks in shared caches have been explored for over a decade. Existing approaches utilize side-channel (non-functional) behaviors such as time, power, and electromagnetic radiation to attack encryption schemes. In this paper, we survey the widely used target encryption algorithms, the common attack techniques, and recent attacks that exploit the features of cache. In particular, we focus on the cache timing attacks against the cloud computing and embedded systems. We also survey existing countermeasures at different abstraction levels.

Keywords Side-channel attack · Cache · Timing

1 Introduction

In the last decade, we witnessed the prevalence of cloud computing and associated platforms by large companies, such as Microsoft, Amazon, IBM, and Google. This trend is due to the convenience, cost savings, and real-time scalability of the cloud. Internet-of-Things (IoT) companies prefer to using commercial cloud services to collect and analyze data, rather than deploying their own infrastructures, considering the costs of procurement and maintenance, and the uncertainty of future server load. For individual users, cloud computing infrastructures such as Amazon EC2 also provide convenient and high-performance machines with reasonable cost. These platforms support different users by virtual machines (VMs) to provide isolation, reduce cost, and maintain utilization. Security and privacy are important design considerations due to the increasing amount of secret and sensitive data in the cloud.

✉ Yangdi Lyu
lvyangdi@ufl.edu

Prabhat Mishra
prabhat@ufl.edu

[1] University of Florida, Gainesville, FL 32611, USA

Cryptographic schemes are heavily used to prevent unanticipated information leakage. For a long time, breaking cryptographic schemes means mathematically cracking the encryption algorithm and inferring the original text from the ciphertext. On the other hand, side-channel attacks are able to extract the secret key without learning the direct relation between plaintext and ciphertext. Once secret key is obtained, deciphering the encrypted information is trivial. The basic approach of finding the secret key is to exploit the implementation of these schemes in real machines and make use of the non-functional information such as power, electromagnetic, and time usage during the process of encryption or decryption. An active area of side-channel attack researches is related to the cache and memory over the last decade with the advantage of high resolution and stability.

Caches are employed between the CPU and main memory (RAM) to compromise the exponentially growing performance gap between them. The overall memory access time is greatly reduced by buffering recently used data. To improve memory access time, modern processors use multiple levels of caches, with the smaller, faster, and more expensive cache in a higher level (closer to the processor). Although caches greatly improve average performance, the time variation between a cache hit and a cache miss has led to many side-channel attack researches during the recent decade. The rationale of these attacks is that accessing cached data in a higher level is more than one order of

magnitude faster than the lower ones, and the attacker can infer partial or full memory access information of the victim by measuring access time if they share cache. In a multi-core system, cache coherence is designed to make sure that every core reads the most recently updated data, for example using MESI, MOSI, and MOESI protocols. Recent cache attacks to multi-core systems take advantage of the inclusiveness of the last-level cache (LLC) to make sure a specific cache line is not present in the other core's private cache.

There are some existing surveys on side-channel attacks. In 2009, Acıiçmez et al. [2] summarized four types of microarchitectural attacks based on the targets: data cache, branch prediction unit, instruction cache, and functional units. However, most of the attacks are targeting single-core systems. A lot of effective attacks on multi-core systems and cloud environments are published in recent years. A recent survey by Ge et al. [22] summarized microarchitectural side-channel attacks and denial-of-service (DoS) attacks with known countermeasures and developed a taxonomy. Compared to [22], this paper describes the pitfalls of encryption algorithms and provides a detailed analysis of the strength and weakness of different attack techniques. With a clear knowledge of how these attacks work, more and better countermeasures can be designed. In Lipp's master thesis [41], cache timing and rowhammer attacks on ARM are summarized together with experimental results of popular attacks. This paper has a broader focus and countermeasures.

This paper summarizes cache and memory side-channel attacks, mainly focusing on cache attacks. This paper is organized as follows. Section 3 analyzes the pitfalls in the implementations of some widely used encryption algorithms, which are heavily exploited by side-channel attacks. In Section 4, we introduce the most commonly used attack methods with their advantages and disadvantages. The next three sections, we summarize the attacks in different platforms with detailed countermeasures in Section 9. Finally, Section 10 summarizes this paper.

2 Background

This section describes the basics on caches and memory hierarchy followed by an overview of side-channel analysis.

2.1 Cache Hierarchy

Cache is used to buffer recently accessed data to exploit spatial and temporal locality. Each cache line holds a number of adjacent bytes in memory. When any byte in the cache line is accessed, a number (determined by the line size) of adjacent bytes are buffered into cache. For a memory access pattern with spatial locality, only the first access triggers a cache miss, while the remaining accesses to successive addresses are all cache hits. The address of memory is decomposed into three parts, namely tag, index, and block offset. The number of bits in block offset part is determined by the cache line size and the block offset determines the relative position inside the cache line. The index determines a set of cache lines that this address can be mapped to. Caches can be categorized by the number of cache lines in one set: direct-mapped caches with one cache line in a set and set-associative caches with multiple choices. Figure 1 shows the basic steps to access one memory location from cache. First, the corresponding set is chosen based on the index part of its address. Then tag comparison is done in parallel for all the blocks in the set. If none of the blocks matches, the request will be sent to the lower-level caches or main memory.

To further decide which cache line to store the data in set-associative caches, replacement policies are applied such as LRU, pseudo-LRU, and round-robin. For example, LRU chooses the least recently used cache line to be replaced when all lines in the corresponding set are occupied. Another important policy of cache is the inclusion policy. It is called inclusive if all data in L1 must also be in L2, and exclusive if the data appears at most in one level of cache. In a multi-core system with inclusive caches, when one core wants to evict a cache line from the processor, it can simply evict it from the last level cache. By inclusive policy, the cache line is guaranteed to be evicted from all the caches including those that are private to other cores. In particular, the Flush+Reload method introduced in Section 4.3 takes advantage of this feature of Intel processor .

The time variation of accessing data from different levels of caches and main memory is very large, which leads to measurable performance difference. If a certain memory address is buffered in the cache, next access to the same or adjacent memory addresses that are mapped to the same cache line will be quick. Otherwise, if the address is not in the cache, next access will be slow. Cache side-channel

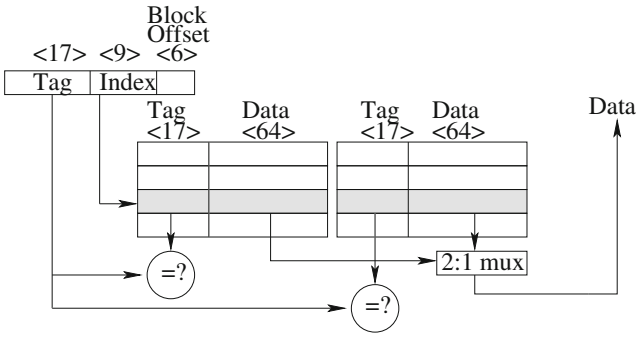

Fig. 1 A 64-KB, two-way set associative cache with 64-byte blocks. The 9-bit index selects among 512 sets. The selected set with two blocks is shown in the figure

attackers make use of this timing variation within the memory hierarchy to learn the memory access pattern of the victim.

2.2 Main Memory

Every process uses its own virtual address to access main memory, and it is limited in its own virtual memory space. Each virtual memory is divided into pages which are mapped to pages in physical memory. The translation from virtual address to physical address is done by the memory management unit (MMU). MMU uses a "page table" as shown in Fig. 2. The size of virtual memory space is decided by the length of address, e.g., the size of virtual memory space for each process is 2^{64} in a 64-bit system. Even with the huge page technique (e.g., 1 GB per page), there are still a large number of pages for a single process. To speed up the translation of recently used pages, a special cache named translation lookaside buffer (TLB) is used. To avoid translation latency for each memory access, the highest level of cache (L1) is typically indexed by virtual address. The lower levels of caches (L2, L3) tend to use part of the physical address as index. As each process only knows the virtual address of its own data, intentionally loading/evicting a specific cache line from L1 is relatively easier than lower level cache that requires the effort to find out the virtual address to physical address mapping.

As the private caches are often large enough to store all data needed by encryption algorithms, cross-core attacks require memory sharing between different cores so that the attacker can learn the memory access pattern of others. Regarding security, one notorious optimization in physical memory is called kernel same-page merging (KSM) or memory deduplication. The goal of KSM is to merge identical memory pages (e.g., unchangeable shared libraries) among different processes or virtual machines to improve physical memory utilization, as shown in Fig. 3. In Linux, it is performed by a kernel daemon named *ksmd*, which periodically scans certain areas of user memory and looks for pages of identical content which can be replaced by a single "copy-on-write" (COW) page [37]. As *ksmd* is transparent to user

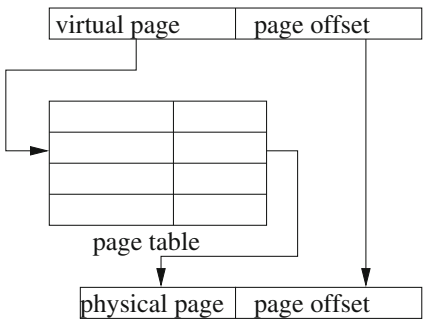

Fig. 2 Virtual address to physical address translation

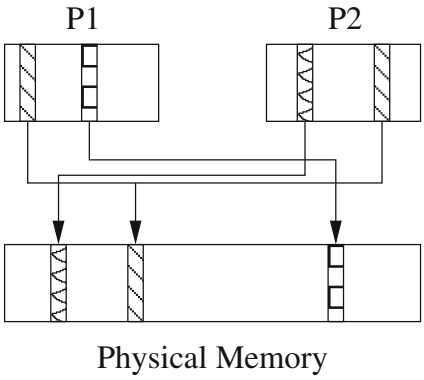

Fig. 3 Memory deduplication: the identical pages in P1's and P2's virtual address space are mapped to the same page in physical memory

processes, it enables different processes to access the same physical memory line using their own virtual addresses. In VMware, memory deduplication is called transparent page sharing (TPS), which continuously scans for identical memory pages using hash functions and then merges them in the physical memory.

2.3 Side-Channel Attacks

Rather than focusing on the mapping between plaintext and ciphertext, a side-channel attack acquires the key by analyzing non-functional behaviors along with encryption or decryption operations, such as time, power, electromagnetic radiation, and so on. Even for theoretically proved strong encryption schemes, it is hard to avoid revealing these physical information. Especially when implementing these algorithms in a real machine, we often impose more requirements such as performance optimization to it, instead of considering merely security. Kocher [39] and Kelsey et al. [36] demonstrated the leakage of sensitive information using cache memory as a side channel. Page [50] expanded this idea and described theoretical attacks via cache misses. The applicability of this approach is limited by the assumption that the cache should be initially empty with respect to the data associated with the algorithm. With these theoretical researches, a lot of side-channel attacks are designed and improved in the recent decade, with fewer samples needed to extract key, and broader applicability from single computer to commercial VM servers such as Amazon EC2, even for the formally verified kernel seL4, side-channel attacks are still effective [17].

3 Common Target Cryptosystems

To transmit an important message through public networks, encryption schemes are required to first encode it into some meaningless text. These encryption schemes have already

been mathematically proved to be hard to break which means a great amount of computational resources are required to decrypt it and get the original message without key. However, theoretical soundness of the cryptographic systems is not enough as many recent attacks target at the real implementations, such as side-channel attacks. In these attacks, information from the hardware (such as power) and architecture (such as cache timing variation) is used to extract key-related information during the process of encryption or decryption. Cache timing attacks have been proved to be effective over the past decade. In the reminder of this section, we briefly analyze the implementation pitfalls of three widely used cryptosystems that are exploited in cache timing attacks.

3.1 RSA

RSA is one of the most widely used asymmetric public-key cryptosystems, which was developed by Rivest, Shamir, and Adleman in 1977. RSA algorithm generates the keys from two randomly chosen large prime numbers p and q. The theoretical strength of the algorithm is the difficulty of factoring the product of two large prime numbers.

One basic operation in RSA is raising a message to the power of the key. For example, during the encryption process, the ciphertext is generated by

$$c \equiv m^e \mod n \tag{1}$$

where $c, m, e,$ and n represent ciphertext, plaintext, key, and the product of p and q, respectively. To implement exponentiation in a real machine, "Square and Multiply" is the easiest algorithm.

Algorithm 1 shows how "Square and Multiply" computes the exponentiation. It scans the bits of the exponent e from left to right. If the current bit of e is one, there would be an additional modular multiplication operation during the current loop. As the sizes of the operands are often very large, this additional operation leads to a measurable computing time variation. Careful measurements and analysis are able to recover the secret key, bit by bit, using statistical analysis.

Algorithm 1 Exponentiation by Square-and-Multiply

$Square - Multiply(b, e, n)$

1: $x = 1$
2: **for** $i = |e| - 1$ downto 0 **do**
3: $\quad x = x^2$
4: $\quad x = x \mod n$
5: $\quad$ **if** $e_i = 1$ **then**
6: $\quad\quad x = xb$
7: $\quad\quad x = x \mod n$
8: Return x

RSA algorithm is expensive compared to other symmetric encryption algorithms. Using it directly to encrypt the message is inefficient. So RSA is often used to encrypt the symmetric key which can encrypt and decrypt the message efficiently. To overcome the expensive computing time, a lot of optimizations are employed:

1. Apply Chinese remainder theory to replace the remainder of the exponentiation divided by n with the remainders of the exponentiations divided by p and q, respectively.
2. Use sliding window to consume multiple bits of the exponent at one step. This optimization uses a set of precomputed multipliers, e.g., $\{a, a^3, a^5, ..., a^{31}\}$, to accelerate modular exponentiation.
3. Use Montgomery modular multiplication [46] to efficiently perform modular multiplication by transforming operands to Montgomery form.
4. Use Karatsuba algorithm to improve multiplication speed when the two operands have the same length.

Although these optimizations greatly speed up RSA algorithm, they also create more opportunities for side-channel attacks.

Percival [53] demonstrated that these optimizations leave "footprint" in the cache and can be utilized by cache side-channel to extract RSA keys in a simultaneous multithreading environment. For example, the multiplier operand can be inferred by cache access time. Due to these optimizations, 200 bits out of each 512-bit exponent (two exponents after applying Chinese remainder theory) can be obtained during "Square and Multiply."

3.2 AES

In contrast to RSA's expensive modular exponentiation, the Advanced Encryption Standard (AES) consists of merely substitution and permutation, which is fast in both software and hardware. Rijndael [20] is the algorithm that has been selected as the candidate for AES by the US National Institute of Standards and Technology (NIST). During AES encryption and decryption, ShiftRow, MixColumn, and SubBytes operations are utilized. To improve encryption and decryption performance, Rijndael's algorithm introduces several precomputed lookup tables to substitute such transformations. For 128-bit keys, 8 lookup tables are precomputed with 256 4-byte words each. They are loaded into memory to quickly encrypt and decrypt by performing simple XOR operations.

In Rijndael's algorithm, the intermediate cipher results are called states. Figure 4 shows a SubBytes operation from [20], which demonstrates the substitution table acting on the state. The initial state is computed by $p_i \oplus k_i$, where p_i and k_i represent the ith byte of the plaintext and the key, respectively. The initial state (the left table in Fig. 4)

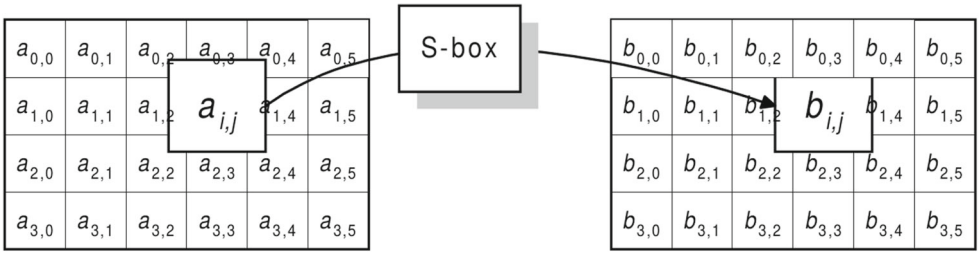

Fig. 4 SubBytes acts on the individual byte of the state [20]. S-box means the substitution table

is used to index precomputed lookup table (S-box in Fig. 4). Then we use the retrieved element to compute state for next round (the right table in Fig. 4). Bernstein [7] exploited timing attacks of the input-dependent table lookup implementations and tested it successfully on many chips including an AMD Athlon, an Intel Pentium III, an Intel Pentium M, an IBM PowerPC RS64 IV, and a Sun UltraSPARC III.

Essentially, timing attacks on fast implementations of AES are based on the key-dependent lookup tables. They use simple XOR operation to quickly transform one state to next. Based on the time to access a certain table element, the attacker can infer whether the element is buffered in cache or not. In other words, the index of lookup table, which is computed by the plaintext, and the key can be learned by another process.

3.3 ECDSA

Elliptic Curve Digital Signature Algorithm (ECDSA) is a digital signature algorithm used by Bitcoin. The signature contains a hash of message to be signed, a randomly chosen cryptographically secure integer, and a private key. One of the operations is called curve point multiplication. A simple implementation could be "double and add," which is similar to "Square and Multiply" in RSA. To avoid the timing variation introduced by the control flow, OpenSSL uses a more secure implementation based on Montgomery ladder, see Algorithm 2.

Algorithm 2 Montgomery ladder point multiplication

$Montgomery - Ladder(d, P)$

1: d represents as binary $d_m d_{m-1} ... d_0$
2: $R_0 = 0$
3: $R_1 = P$
4: **for** $i = m$ downto 0 **do**
5: **if** $d_i = 0$ **then**
6: $R_1 = add(R_0, R_1)$
7: $R_0 = double(R_0)$
8: **else**
9: $R_0 = add(R_0, R_1)$
10: $R_1 = double(R_1)$
11: Return R_0

Although Algorithm 2 eliminates the total time variation introduced by control flow, Yarom and Benger [69] are still able to recover key by using Flush+Reload. Their method first flushes both instructions in line 6 and line 9 in Algorithm 2, and then reloads them to decide which branch is actually taken by analyzing the timing variation.

4 General Methods for Side-Channel Attacks

Cache side-channel attacks can be categorized into two classes, as shown in Table 1. Please note that the overhead may not be comparable since different platforms, optimizations, environment noises, and assumptions are used. Also, offline analysis overhead is not shown in the table.

1. Time-driven: In time-driven attacks, the attacker measures the total execution time of cryptographic operations to extract sensitive information [4, 7, 14, 39, 65]. The rationale behind the attack is that the execution time varies with the execution paths or cache hits/misses, which is often key-related. Therefore, the attacker can extract keys by controlling the contents in the shared cache and measuring the running time of the victim. However, as the time-driven attacks measure the whole coarse execution time, they suffer from the noise of OS and network. Thus, a large number of samples are needed to apply a statistical evaluation to extract key-related information. Better time-driven attacks mean less encryption samples to extract key. The main advantage of this attack is the wide applicability which only requires execution time measurement.
2. Access-driven: In access-driven attack, the attackers monitor whether a specific component is used by the cryptographic operations or not, including data cache [26, 53, 59], instruction cache [1], and branch prediction cache [5]. This information is learned by measuring the time of accessing this component from memory. If it has been accessed by the victim, the attacker should monitor a cache hit, otherwise a cache miss. One common access-driven attack is monitoring the data cache to learn which lookup table entries are being used when performing AES.

Table 1 Performance and overhead of cache timing side-channel attacks

Type	Paper	Platform	Target	Extracted information	Overhead
Time-driven	[48, 59]	Athlon 64	AES (OpenSSL 0.9.8)	Full	500,000 encryptions
	[29]	Intel i7-870	Address space layout randomization (ASLR)	The address of the syscall handler	180 probing
	[65]	Cortex-A8, L4 microkernel	AES (Barreto)	Limit to 4 choices per key byte	1,600,000 samples
	[11]	ARM9	AES (OpenSSL 0.9.8k)	Remaining key space $2^{43.8}$	2,660,000 measurements
	[32]	Intel i5-3320, VMware and Xen	AES (OpenSSL 1.0.1f)	30–40 bits	2^{30} encryptions
Access-driven	[48, 59]	Pentium 4E	AES (OpenSSL 0.9.8)	Full	16,000 encryptions
	[26]	Pentium M	AES (OpenSSL 0.9.8n)	Full	Training in own machine 168,000 encryptions, 100 encryptions to extract key
	[43]	Intel Xeon E5 2690 Xen 4.4, Intel Core i5-3470 VMware ESXi 5.1	ElGamal (GnuPG 1.4.13 and 1.4.18)	Full	79,900 observed exponentiations
	[47]	Intel Core i7, Safari	Browser activity	82.1% accuracy	5000-element vector
	[75]	Intel Core 2 Q9650, Xen hypervisor	GnuPG 2.0.19, libgcrypt 1.5.0	98.1% accuracy	140,000 operations, 300,000,000 Prime+Probe trials
	[34]	Intel i5-650, Xen 4.1 and VMware ESXI 5.5	AES (OpenSSL 1.0.1f)	Full	650 encryptions
	[33]	Intel i5-3320, VMware ESXI 5.5	AES (OpenSSL 1.0.1f)	Full	400 encryptions
	[31]	Intel Xeon E5-2670	RSA (libgcrypt 1.6.2)	Full	64,000 decryptions
	[70]	Intel i5-3470, VMware ESXi 5.1	RSA (GnuPG 1.4.13)	Average 96.7% of the bits	A single decryption
	[69]	HP Elite 8300	ECDSA (OpenSSL 1.0.1e)	571 bits	1 signing process
	[27]	Intel i5-2430M, VMware ESXI 5.5	AES (OpenSSL 1.0.1g)	Full	30,000 encryptions

Note that both time-driven attacks and access-driven attacks rely on measuring time information. The key difference in timing-driven versus access-driven is that in the former case, the attacker has to measure the victim process' execution time, while in the latter case, the attacker measures the execution time of an operation of its own. This difference gives access-driven attacks higher fidelity than time-driven attacks. One important requirement of these two kinds of attacks is that the attacker process should be allowed to run in the same processor with the victim process, so that they can share caches with each other. There are many methods to conduct cache side-channel attacks. In the subsequent sections, only some of the commonly used methods are listed.

4.1 Evict+Time

Evict+Time [48] is a time-driven attack which learns information from the execution time of the victim process. It consists of three stages:

1. Trigger the victim process.
2. (*Evict*) Fill specific cache set with attacker's data; hopefully, it can evict the buffered data of the victim.
3. (*Time*) Measure the execution time of victim process again.

After the first execution of the victim process, the data used for encryption or decryption is buffered in the cache, such as some elements of lookup tables in AES. If the attacker luckily evicts these cache lines, the second execution of the victim process will be slower. On the other hand, if the evicted cache lines are not useful, the second execution will be faster. Therefore, the timing information of the second execution reveals the memory access pattern of the victim process. For AES, as the indexes of lookup tables are computed by the private key, the timing information reveals part of the key.

There are a few weaknesses of this method. First, the assumptions of this method are strong. We assume knowledge of the (virtual) memory address of useful data (such as each lookup table of AES), or at least the mapping from virtual address to physical address, and the ability to trigger an encryption and control when it begins and ends. Second, measuring the time of encryption process is imprecise. Osvik et al. [48] introduced this method and used it to extract full AES keys from an artificial services using OpenSSL library calls. They also pointed out the weakness of this method that additional codes are executed when triggering the encryption, so the timing contains considerable noise from scheduling, page table misses, and other sources. Third, the attack is slow. Since there is only one set evicted each round, more time is needed to fully recover the key.

4.2 Prime+Probe

Prime+Probe [48, 53] first fills the cache with its own data and then checks which one is evicted after triggering the encryption process.

1. (*Prime*) The attacker occupies specific (or all) cache sets with data.
2. The victim process is scheduled to run and access its own data.
3. (*Probe*) The attacker accesses the same data which is loaded into cache during the Prime stage. If the victim process has loaded some data that maps to the same cache lines and evicts the data of the attacker, a longer probe time will be observed by the attacker due to a cache miss. Otherwise, the data would be in the cache and the probe time is relatively short.

Rather than measuring the time of encryption process as Evict+Time, in Prime+Probe, the attacker only measures its own running time which is effective and noise-resistant. Furthermore, as different cache sets can be inspected simultaneously during probing stage, less encryption calls are required.

The mapping between virtual and physical addresses makes the attack complicated. For a virtually indexed cache so as most modern L1 cache, it is not a problem. For a physically indexed cache, we need to find out the mapping between virtual address and physically address first. One benefit from memory deduplication is that they can use different virtual addresses to refer to the same physical address when the attacker and victim use the same shared library.

4.3 Flush+Reload

Gullasch et al. [26] first came up with the main idea of this method and attacked L1 cache on AES. Yarom et al. [70] extended his method to L3 cache for cross-core attacks and named it Flush+Reload. It is a powerful attack which determines a specific instruction or data accessed by victim process.

1. (*Flush*) Flush a memory line from the cache.
2. The attacker waits for the victim process to run.
3. (*Reload*) Measure the time to reload the memory line.

In the paper of Yarom et al. [70], they introduced an attack on the instruction cache. Their method flushes and measures the time to reload the same instructions in the Square, Multiply, and Reduce functions. If the victim process executes these lines before reloading, reloading is faster than the other case. So the timing variation of reloading reveals the exact execution path of the victim process.

It is easy to see that Flush+Reload is actually a variant of Prime+Probe. There are two differences between them. First, Flush+Reload relies on memory deduplication to share pages between the attacker and the victim processes. So it typically works on read-only shared memory. Second, the data loaded in the first step is different. In Prime+Probe, the attacker loads any data that can be mapped to the monitored cache set. As the data with the same index in their address maps to the same cache set, there are a lot of candidates to choose from. For such attacks on cache with large associativity such as the last level cache, the Prime stage requires to load multiple cache to fill the whole set. However, in Flush+Reload, the attacker flushes out some specific memory to fill the whole set from the cache, by which a high resolution is achieved.

The magic behind Flush+Reload is the instruction *clflush* that is able to evict specific memory lines from all the cache levels, including the shared last level cache. This attack can only work in such processors with inclusive last-level caches to achieve cross-core evicting, i.e., when one core evicts the data from shared last-level cache, the data is also evicted from all the private caches of other cores.

One important problem in the attack is to decide how long should the attacker wait for the victim process in the second stage, since the attacker does not have any control over the victim process. If the attacker waits for a too short period of time, the victim may not reach a certain instruction yet, which leads to false-negative conclusion. Other noises come from prefetching and speculative execution.

4.4 Other Variations

As Flush+Reload relies on the x86 instruction *clflush*, it is easy to design a system which restricts the usage of *clflush*. Gruss et al. [24] replaced the flush process in Flush+Reload with eviction and proposed Evict+Reload technique. The eviction is similar to Evict+Time which evicts certain memory from cache by accessing addresses that are mapped to the same cache set. The disadvantage of this attack is the prior knowledge of virtual to physical address mapping.

In 2016, Gruss et al. [25] came up with Flush+Flush after observing that the execution time of *clflush* instruction also depends on whether the memory line is in the cache or not. If the memory line has been loaded into cache by the victim, it takes longer time to complete *clflush*. Replacing Reload with Flush can reduce the number of cache misses incurred by reload and will not trigger prefetching, which can help this attack evade some detection mechanisms.

5 Cache Attacks on Single-Core Systems

Processes that run in the same core share L1 data cache, L1 instruction cache, and branch prediction cache. This level of sharing enables the possibility of cross-process attacks. Tsunoo et al. [60] exploited collisions in the memory lookups invoked internally by the cipher instead of collisions created by the attacker, and demonstrated a timing attack using this knowledge. Brumley and Boneh [14] demonstrated a more advanced and practical timing attack against RSA to an OpenSSL-based web server, which was improved by [3]. Percival [53] demonstrated how to apply an attack to RSA by monitoring the multipliers in cache. To defend this attack, Intel recommends crypto-algorithms with no secret- or data-dependent memory access pattern at coarser than cache line granularity [12]. Scatter-gather technique, which scatters each multiplier across cache lines and gathers them to a single buffer before multiplication, ensures that secret-dependent accesses are at a finer than cache line granularity to defend side-channel attacks. However, Yarom et al. [71] recently proposed an attack exploiting cache-bank conflicts to bypass the scatter-gather technique in OpenSSL. They recovered 4096-bit RSA with 16,000 decryptions showing that the offsets of multiplier accesses inside cache line still depend on the key.

In 2005, Bernstein [7] proposed a practical time-driven attack against the OpenSSL implementation of AES. His main idea is to exploit the statistical patterns in the encryption time of different plaintexts P under a known key K. After learning the time information from the interested encryption process, key candidates are chosen based on the correlation followed by an exhaustive search. As the method only needs time measurement, it is simple and portable. However, there are some disadvantages in this attack.

1. Privileges are required to perform cryptographic operations with a user-defined key, which is the critical part of this attack to learn the correlation.
2. As discussed in Section 4, time-driven attacks suffer from noise in a real system. A large number of samples are required to extract key.

Independently from Bernstein, Osvik et al. [48] describe a similar attack named Evict+Time (see Section 4.1). They also proposed an access-driven attack named Prime+Probe (see Section 4.2), to AES with the knowledge of physical and virtual addresses of lookup tables. Acıiçmez et al. [4] exploited the internal collisions of AES and described a new cache timing attack. Their method can work remotely under a multitasking or simultaneous multithreading system.

Gullasch et al. [26] presented an asynchronous model of cache side-channel attack, which is the original idea of Flush+Reload on AES. In their method, the attacker is assumed to have an identical machine with the victim. Two neural networks are trained in this machine to carry out offline learning to extract key from 168,000 encryptions. After the training, the attacker does not need to know plaintext or ciphertext, and only 100 encryptions are enough to extract the key. As the neural network is trained in a particular machine (Linux OS, single-core x86 system), substantial adjustments are required after updates of the operating system or moving to a different machine.

Single-core attacks mainly focus on the first-level cache. There are benefits to attack L1, such as less load instructions are needed to prime or evict the cache due to the small size of L1. However, as the access time variation between L1 caches and L2 caches is only a few cycles in modern processors, the noise of time measurement in the real system makes the attack harder.

6 Cache Attacks on Multi-core Systems

Modern chip multiprocessors (CMPs) use shared cache, such as last-level cache, to reduce communication latency between threads or processes. Cache side-channel attacks make use of LLC to generate contention as long as two cores are in the same package. With the help of cache policies, such as replacement policy, inclusive policy, and so on, one process can intentionally remove data from another core's private cache.

Based on Prime+Probe technique, Liu et al. [43] demonstrated a cross-core, cross-VM attack on "Square and Multiply" exponentiation algorithm and sliding-window exponentiation introduced in Section 3.1. They pointed out a few problems to handle before constructing efficient Prime+Probe on LLC:

1. As L1 and L2 are usually large enough to satisfy memory accesses, the encryption process rarely touches LLC. So merely observing LLC is not enough to gather the memory access pattern of another core. Similar to Flush+Reload, cache inclusiveness is important to ensure the visibility of cross-core memory access pattern.
2. When conducting Prime+Probe in L1, the whole cache is primed and probed. However, as LLC is greatly larger than the higher levels, it is infeasible to prime and probe the whole cache in each round. They proposed to identify relevant security-critical accesses by scanning the whole LLC and looking for temporal access patterns.
3. As discussed in Section 4.2, the first step of Prime+Probe is to occupy specific cache sets. To achieve that, an eviction set is constructed in the virtual space of the attacker. However, as LLC is physically indexed, mapping from virtual address to physical address is required.
4. Probing resolution is another consideration. Priming one cache set in LLC requires more load instructions, since LLC has higher associativity. Furthermore, longer latency of LLC leads to longer probing time. They observed that probing an LLC is about one order of magnitude slower [43].

After dealing with these problems, they repeatedly decrypted a known plaintext using the same key in GnuPG. They claimed that their attack can work in a real cloud environment due to their minimal assumptions, such as cache inclusiveness, large-page mappings. However, as the real cloud environment contains more noise and other mechanisms to prevent such attack, there are no experiments in their paper to show it is practical to attack a real cloud.

Oren et al. [47] extended Liu et al. [43] to web-based cache side-channel attack without the assumption of large-page. As long as the victim launches the attack from sandboxed JavaScript, the browsing activity is monitored. By targeting at web, the attack can be applied to different platforms, such as Mac OS, Linux, and cross-VM, and does not require the victim to install program from the attacker.

7 Cross-VM Level Attacks

With the prevalence of cloud computing, security risks over the cloud are also rising. Cloud computing infrastructures, such as Microsoft Azure and Amazon EC2, allow multiple users to share physical machines rather than dedicating one machine to one user. Users can run their own virtual machines (VMs) to be isolated with other users. The layer between virtual machine and physical machine is called virtual machine manager (VMM) enabling personal configurations and managing resources.

7.1 Co-resident

To apply cache side-channel attacks to cross VMs over the Internet, one important challenge is to physically mount the attacker's VM to be co-resident with the target VM, while recent cloud infrastructures try to distribute tenants over all physical resources. Ristenpart et al. [55] were the first to introduce a method to mount the attacker to be co-resident with the target so that LLC attack can be applied

on Amazon's EC2. As Amazon's EC2 allows customers to instantiate VMs on demand, Ristenpart et al. repeatedly instantiated new VMs in an empirical way to maximize the likelihood of placing co-resident with the target. During this period, a simple and low-overhead method is applied to check whether the target and the victim are co-resident. Although the information revealed in their attack is quite coarse, such as cache usage and traffic rate estimation, their co-residency technique provides a way to let cache side-channel attacks break through the sandboxing and leak cryptographic information.

Even in 2016, it was not hard to get co-resident in cloud settings. For example, İnci et al. [31] launched 4 accounts in Amazon EC2 and launched 20 instances for each account, then performed LLC co-location detection test to determine co-location pairs. They successfully got 7 pairs among the 80 instances. Other than co-residency, Owens and Wang [49] observed the special property of the delay of reading and writing to a deduplicated memory. They proposed an OS fingerprinting mechanism by accessing the unique pages of different OS to decide the OS type and version of other VMs, which is helpful to subsequent attacks. With the help of co-resident mounting, more and more researches start to attack cloud environments by extending and applying similar techniques used in local multi-core settings.

7.2 Prime+Probe

Zhang et al. [75] demonstrated an instruction cache-based access-driven side-channel attack on Xen virtualization platform. They first assumed the knowledge of the process running in the victim VM and the ability to have a copy of it. Their main steps are shown in Fig. 5, including the preprocess step to mount the VM to be co-resident with the victim and Phase 3 to reduce background noise. Their attack is against libgcrypt v.1.5.0 cryptographic library, in particular the "Square and Multiply" implementation which is commonly used in RSA and ElGamal (see Section 3.1). As their attack is against L1 instruction cache, the attacker's virtual CPU needs to frequently regain control the same core with the victim, which is relying on a weakness in the Xen scheduler. They performed 300,000,000 trials and 6 h to collect data.

Irazoqui et al. [34] proposed a new cross-core cross-VM attack that does not rely on memory deduplication. However, their cross-VM attack relies on hugepages of LLC, i.e., each page with 2-G size. The benefit of hugepages is to reduce table entries in MMU, while reveals 21 bits of address information rather than 12 bits in page size 4 K from the security perspective. Their attack is a variation of Prime+Probe by exploiting access time variations from the OpenSSL implementation of AES. Their attack is very efficient compared with Flush+Reload due to the small number of sets being profiled. Similarly, İnci et al. [31] expanded the Prime+Probe method in [43] based on hugepages and profiled a small number of sets in LLC to extract RSA key from Amazon EC2.

7.3 Flush+Reload

The Flush+Reload is one of the most effective, high-resolution cache side-channel attacks. Yarom et al. [70] applied this method both on local multi-core system and on cross-VM scenarios such as VMware ESXi 5.1 and CentOS 6.5 with KVM. The details of this method are introduced in Section 4.3. In summary, this attack used *mmap* to gain a copy of victim's executable file. Relying on page de-duplication mechanisms, they used *clflush* in x86 to flush a certain instruction out of all levels of cache to monitor the code path of the victim. Their attack is also against "Square and Multiply" in RSA whose execution path is determined by the bits of key. They first flushed out one instruction line out of memory, then probed whether the victim used this instruction or not. This high-resolution attack is able to recover 96.7% of the bits by a single signature. Yarom et al. [69] also showed Flush+Reload can be used to recover the secret key from the Elliptic Curve Digital Signature Algorithm (ECDSA). With the secret key around 571 bits, the attack took only one signing process and less than 1 s to recover the key.

Flush+Reload technique can also be used to recover key from the implementation of AES in OpenSSL in VMware [33]. Irazoqui et al. assumed the knowledge of the offset of the lookup tables with respect to the library. As VMware provide transparent page sharing, they are able to recover key across cores in VM within less than 1 min

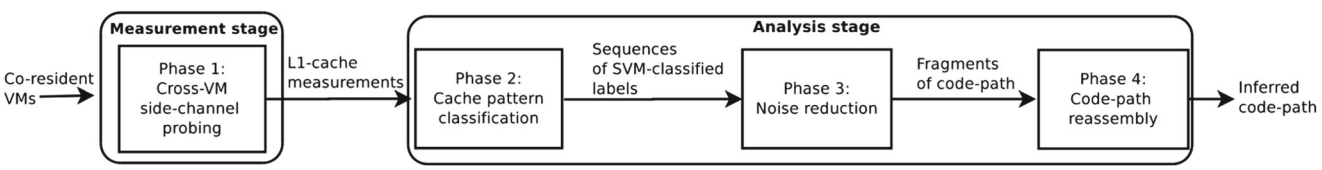

Fig. 5 Main steps in the attack from Zhang et al. [75]

by monitoring a single-shared memory line. Before [33], Irazoqui et al. [32] also showed how to use Bernstein's timing side channel to extract key-related information from AES implementation running in VMs such as Xen and VMware. The difference between these two works is the efficiency: the latter takes 2^{29} encryptions and more than 4 h while the former takes only 1 min.

Other Flush+Reload approaches include a known-ciphertext cross-VM attack by Gülmezoğlu et al. [27] against AES, and an automation-driven framework by Zhang et al. [76] to attack the web applications in PaaS environments.

8 Embedded and Mobile Devices

Embedded and mobile devices include smartphones, tablets, wearables, and automobiles. With billions of such devices in daily life, their security problem is becoming more and more important. There are many successful applications of side-channel attacks on embedded and mobile devices, such as power side-channel attacks and electromagnetic side-channel attacks. Cache side-channel attacks have been extensively studied on x86 architectures, but much less so on embedded devices.

8.1 Unique Characteristics in Embedded and Mobile Devices

Embedded and mobile devices have four unique characteristics that affect the success of side-channel attack on them. First, as the processors of embedded devices are often designed to support limited applications, their instruction sets are based on RISC architecture, such as the most widely used ARM processor. These instructions are not so "powerful" as x86; for instance, the lack of similar instructions as *clflush* to flush out specific memory out of all cache levels makes Flush+Reload (see Section 4) unable to be applied directly. Then, the sizes and levels of caches in embedded systems are typically smaller. However, in recent years, the processors are becoming more and more powerful, and cache levels are increasing. Nowadays, the most high-performance embedded processors contain two (Cortex A73) or three levels (Apple A9) of caches. Next, the details of the cache are poorly documented, such as cache coherence protocols, which makes it hard for the attacker to be successful. Finally, the implementations of encryption schemes differ from that in desktop or server in some aspects such as the size of lookup table.

In the next three sections, we review existing cache side-channel attacks against embedded systems in two broad categories.

8.2 Time-Driven

Cache timing channel attack is explored in the area of ARM-based devices since 2010 by Bogdanov et al. [11]. They proposed a new cache timing attack, namely differential cache-collision attack, to the OpenSSL implementation of AES running on ARM9 microprocessors. It is called a wide collision, if the same AES S-box value is queried twice for a plaintext pair. The authors tried to trigger wide collisions by choosing plaintexts P_1 and P_2 in a specific way where two plaintexts only differ at diagonal elements in the 4×4 matrix representation (see Fig. 6), then sent (P_1, P_2) to the encryption routine and measured the encryption time of P_2. If a wide collision happened, the encryption time is expected to be short. After trying a large number of (P_1, P_2) pairs, key recovery method is applied on those pairs that triggered wide collisions. One problem of this method is a large number of false positives. It is resolved by increasing the number of computations, which also increases the complexity of the key search phase.

The applicability of Bernstein's cache timing attack [7] in ARM processors is investigated by [56, 57, 65]. Weiß et al. [65] demonstrated that cache timing attacks can bypass system virtualization and compared the vulnerability of different AES implementations.

With respect to the cache-collision attack and Bernstein's cache timing attack, Spreitzer and Plos [57] investigated their applicability in real environments: an Acer Iconia A510, a Google Nexus S, and a Samsung Galaxy SIII. The common problem of these two methods is the key search space. Differential cache-collision attack reduces key space from 128 to 52 bits and Bernstein's cache timing attack reduces key space to 58–73 bits, which is still too large

$$P_1 = \begin{bmatrix} a_0 & x_4 & x_8 & x_{12} \\ x_1 & a_1 & x_9 & x_{13} \\ x_2 & x_6 & a_2 & x_{14} \\ x_3 & x_7 & x_{11} & a_3 \end{bmatrix} \quad P_2 = \begin{bmatrix} e_0 & x_4 & x_8 & x_{12} \\ x_1 & e_1 & x_9 & x_{13} \\ x_2 & x_6 & e_2 & x_{14} \\ x_3 & x_7 & x_{11} & e_3 \end{bmatrix}$$

Fig. 6 Pairs of 16-byte plaintexts represented in 4×4 AES state [11]

for exhaustive key search. Another problem of differential cache-collision attack is that encryptions with or without wide collisions are hardly distinguishable.

8.3 Access-Driven

L2 cache in ARM is physically indexed, which forms the basis for the powerful and high-resolution access-driven attack Flush+Reload. However, before applying Flush+Reload to ARM processor, a few problems [72] need to be tackled:

- How to realize *clflush* in ARM to flush out a specific memory from all levels of caches?
- How to get high-resolution clock which is accessible in x86 as *rdtsc*?
- What is the cache coherence of ARM? Flush+Reload asks for strict inclusiveness to apply cross-core attacks.

Zhang et al. [72] proposed a return-oriented Flush+Reload attack on last-level caches of ARM processors to detect hardware events and trace software execution paths. They tackled the above problems by (1) a processor-specific cache-flush interface taking advantage of clearcache system call, (2) POSIX *clock_gettime* system call, and (3) a method using only cache timing to empirically determine whether the L2 cache is inclusive, exclusive, or non-inclusive.

Lipp et al. [42] showed that Prime+Probe and Evict+Reload attacks can be applied to Android smartphones without root privileges to recover keys. Furthermore, their attack techniques can be used to monitor keystroke and swipe actions that require a high resolution and high accuracy. To get the mapping of virtual-to-physical translation, they took advantage of a vulnerability (reading /proc/<pid>/pagemaps) in the Android kernel.

9 Countermeasures

There are a wide variety of countermeasures against cache timing attacks. Table 2 broadly classifies them into three categories and lists their overhead. This section describes attack detection methods followed by three classes of effective countermeasures.

9.1 Attack Detection

Attack detection enables users to realize the existence of an attacker and take proper countermeasures, such as relocating virtual machine. Once a side-channel attack is detected, a suitable countermeasure can be employed. Side-channel attacks typically cause abnormal cache references and misses. So hardware performance counters can be used to detect cache attacks. Hardware performance counters are common in modern microprocessors, with special purpose registers to store specific program behaviors such as clock cycles, cache hits/misses, branch misses, and so on. Chiappetta et al. [15] proposed three methods based on hardware performance counters to detect Flush+Reload methods by Yarom et al. [70]. Machine learning and neural networks are used in two of their methods to deal with the potential presence of false positives and increase the confidence of the detection. In the training network, they used total instructions, total CPU cycles, L2 cache hits, L3 cache misses, and L3 cache total accesses as input features. Their method is able to detect Flush+Reload attacks in about one fifth of the total time needed by the attacker, with no requirements to modify hardware or operating system. Payer [52] developed their detection system HexPADS by collecting hardware performance counters to analyze abnormal behaviors by side-channel attacks. The metrics to define attacks include cache behaviors, execution time, loaded libraries, and so on.

In cloud environments, as physical co-location [55] is the first step in the cloud setting side-channel attack such as Prime+Probe and Flush+Reload, the techniques to detect such co-location can be crucial to prevent such attacks. Zhang et al. [74] proposed a method that inverts cache side channel for tenants to verify physical isolation of their VMs. Bates et al. [6] utilized traffic analysis to determine co-location in cloud. Zhang et al. [76] demonstrated that de-duplication enables co-location detection from co-located VMs in PaaS clouds. Wu et al. [66], Zhang et al. [68], and Varadarajan et al. [62] showed that memory bus contention can be used to detect co-location. In 2016, İnci et al. [30] did experiments on three famous commercial clouds, Amazon EC2, Google Compute Engine, and Microsoft Azure, then compared three co-location detection methods. The results show that co-location in these cloud services is still possible.

9.2 Code-Level Countermeasure

This section describes two types of code-level countermeasures.

9.2.1 Constant-Time Techniques

Most cache side-channel attacks are based on the the variation of encryption time related to the key and data. The time variation can come from memory accesses (e.g., AES in Section 3.2) and branches (e.g., RSA in Section 3.1). Constant-time techniques are used in some cryptographic libraries (e.g., NaCl by Bernstein et al. [8]) to prevent cache side-channel attacks. There are some drawbacks in this method:

- Constant-time techniques are difficult to implement due to hardware complexity, especially with the goal to

Table 2 Categories of countermeasures against cache timing attacks, and their overhead

Category	Technique	Paper	Method details	Performance/overhead
Code	Constant-time	[8]	NaCl	More than 80,000 crypto_box operations per second in AMD Phenom II X6 1100T
		[35]	AES using bitslicing	Improves performance to 6.92 cycles/byte, compared to original 10 cycles/byte
		[28]	AES using vector permute instructions	150% improvement on the G4e, 41% slower compared to [35] on x86-64
	Compiler	[18]	Eliminate key-dependent control flow	Slowdown of 24.0 (maximum), code size increases by 3.21 times (maximum)
		[19]	Randomly choosing different execution path	Slowdown varies from 1.25 to 2.1 using 10–50% insertion
Operating system	Restrict fine-grained time measurements	[63]	Modify the value of *rdtsc* read by guest operating system	Performance unchanged when 4096 cycles are fuzzed by $2\mu s$
		[44]	Add a real delay to $rdtsc$	Slowdown of 1.29 (maximum)
	Prevent physical memory sharing	[77]	Copy-on-access memory management system	The response time of Apache 24.7 web server increases by 20%, 25% throughput degradation, 2–15% overhead for requests
Architecture	Cache flushing	[73]	Repeatedly clean L1 cache	7% performance overhead
		[23]	Flush all cache levels during context switch	15% performance overhead
		[61]	Minimum runtime guarantee (MRT) by reducing preemption frequency	Median latency increases between 10 and 50 μs for 5ms-MRT
	Cache partition	[51]	Strided cache lines, adaptive line sizes, a mask	Not available
		[64]	Dynamically lock cache lines	Less than 2% on average for SPEC2000 benchmark
		[38]	Lock stealth memory in cache	5.9% overhead for SPEC2006
		[54]	Cache partition using page coloring	Slowdown of 3.6 (maximum) with 50% coloring
		[23]	Selective cache flushing and cache coloring	15% overhead for Apache benchmark
	Randomization	[64]	Randomize cache mappings	1% overhead for SPEC2000 benchmark
		[40]	Extend random permutation cache to allow critical process to gain control	63% overhead (maximum) for AES

achieve high performance at the same time, for some time-consuming algorithms such as RSA.
– Constant-time implementations are platform-dependent. For example, the "constant-time" fix in OpenSSL against Lucky Thirteen attack still exhibits data-dependent execution time on ARM [17].

Brickell et al. [13] revisited AES implementation and proposed an alternative way to use a compact S-box table small enough to be able to prefetch all elements into only 4 cache lines, and randomly permute tables frequently. By using x86 SSE SIMD instructions, the critical part of their permuted compact round is constant time without branches. The experimental results show that the distribution of their average execution time follows a Gaussian distribution.

Bitslicing is a technique to achieve efficient cryptographic algorithms [10]. Matsui and Nakajima [45] showed the constant-time implementation of AES using bitslicing can achieve performance improvement. Käsper and Schwabe [35] presented an efficient constant-time AES implementation using bitslicing. Their implementation can achieve 6.92 cycles/byte on Intel Core i7, compared to 10 cycles/byte on the same platform using lookup table-based implementation of AES. Hamburg [28] proposed another efficient way to eliminate data- and key-dependent branches and memory references using vector permute instructions.

9.2.2 Compiler Techniques

Modifying encryption implementations by adding noise or randomization is a potential method to defend side-channel attacks. One example of eliminating timing variation is adding dummy operations to weaken timing signals [48, 50]. However, to completely remove all side-channel effects from encryption implementation requires a lot of manual programmer effort with the risk of being incompatible in different platforms. To overcome this disadvantage, compiler techniques have been proposed to automatically change implementations.

Coppens et al. [18] demonstrated that automated compiler techniques can be used to defend side-channel attacks. They eliminated key-dependent control flow by eliminating the conditional move instructions in a compiler back-end using if-conversion. The elimination of control flow can also indirectly defend side-channel attacks against instruction cache and branch prediction. Cleemput et al. [16] evaluated several compiler techniques to reduce time variations caused by data flow. Their results show that there is a trade-off between security and performance using compiler techniques.

Crane et al. [19] thwarted cache-based side-channel attacks by randomly choosing different execution paths during runtime. Different paths are generated by NOP insertion, function reordering, register randomization, and instruction substitution to ensure the semantical equivalence, which results in infinite number of paths theoretically. These functional equivalent copies of execution paths are randomly chosen during runtime to generate exponentially different results of execution time.

9.3 Operating System-Level Countermeasures

This section describes three types of OS-level countermeasures.

9.3.1 Restricting Fine-Grained Time Measurements

Osvik et al. [48] suggested to hide timing information such as adding random delays or normalizing all timings to a fixed value. They also noticed the implementation difficulties and performance overhead by this approach. As many timing side-channel attacks use *rdtsc* to obtain timing information, Percival [53] suggested disabling the use of *rdtsc* or more practically limiting the frequency of reading time stamp counter. Vattikonda et al. [63] weakened timing channels by implementing "fuzzy time" in virtual machine manager. They modified the value of the *rdtsc* register taking advantage of the *softtsc* kernel option in Xen.

Martin et al. [44] identified three ways for the attacker to gather timing information:

(a) Internal sources in hardware, such as time stamp counter.
(b) External sources come from other computers or devices. Martin et al. [44] claimed that external sources are not fine enough to distinguish microarchitectural events.
(c) Virtual clocks created by software. Percival [53] proposed a virtual clock implementation on multiprocessor system with shared memory. In his implementation, one thread repeatedly increments a memory location which is used by another thread as a time counter.

With respect to hardware sources, Martin et al. [44] revisited some techniques such as disallowing user-space *rdtsc* instruction, masking the least significant bits, or adding random offset. The results show that these techniques are either impractical or insufficient. They proposed a solution as adding a real delay to *rdtsc* calls to limit the frequency. For software clocks, they used a detector to detect shared memory communications and a delay producer to insert delays to these communications. They validated the correctness of their implementation without breaking the existing software. Then they proved that their approach can defend statistical analysis which is a concern in [48, 53] as a potential way to defeat *rdtsc* fuzzing.

9.3.2 Preventing Physical Memory Sharing

Cross-core attack Flush+Reload and its variations against LLC rely on physical memory sharing between different processes to leak information. VMware has turned off transparent page sharing by default. Zhou et al. [77] proposed a copy-on-access memory management subsystem named CacheBar to prevent physical memory sharing between containers. They define physical page state transitions as in Fig. 7. This scheme automatically creates a copy of the physical page demanded by another security domain. They used model checking to formally prove the correctness of copy-on-access and experimental results to show the low overhead.

9.3.3 Cache Flushing

Cache flushing during context switch can be used to defend side-channel attacks based on L1, BTB, and TLB, which are relatively small to be flushed during context switch. Zhang and Reiter [73] proposed a periodic cache cleansing mechanism to mitigate side-channel attacks. By repeatedly cleaning L1 cache, this approach effectively eliminates timing variation exploited by the attacker. Extensions to other resources are discussed, such as branch prediction cache. They used two modes and skipped unnecessary cache cleansings to achieve less than 7% performance overhead. However, users should be involved in defining which operations are sensitive to trigger cleansing. Godfrey and Zulkernine [23] suggested flushing all cache levels during context switch in VM scheduler. A new field is added to VCPU to indicate the owner of current cache data. Switching to idle or the same domain will not trigger cache flushing. They showed that their hypervisors can effectively prevent side channels with less than 15% overhead. Varadarajan et al. [61] are the first to propose that increasing a minimum runtime guarantee can mitigate side-channel attacks by reducing preemption frequency. Then they integrated a state-cleansing mechanism for L1 cache and branch predictor, and measured 8.4μs overhead from the stand-alone cleansing.

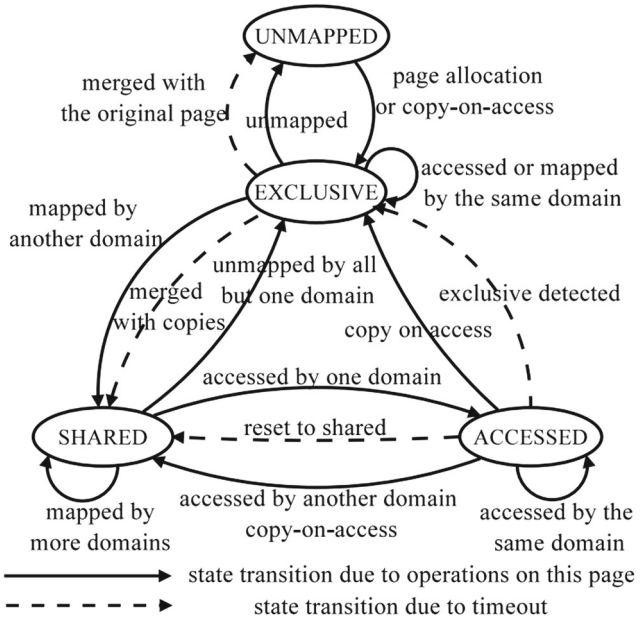

Fig. 7 State transition of a physical page from Zhou et al. [77]

9.4 Architectural Level

Most side-channel attacks described above require sharing cache between processes or cores. By repeatedly writing and reading in the shared cache, the attacker is able to learn the memory access pattern of the victim. So restricting the ability from architectural level can be an effective way to prevent side-channel attacks.

9.4.1 Cache Partition

Cache coloring is proposed as an effective mechanism to defend parallel side channels, which has already been heavily exploited to improve the performance by avoiding excessive cache conflicts. Cache is divided into several groups when using cache coloring, and some specific fraction of memory address is used to decide which group to map the data. Percival [53] suggested to avoid cache sharing or selectively evicting cache based on thread. Page [51] proposed cache partitioning to block cache-based side-channel attacks with high design and performance cost. Wang and Lee [64] proposed the PartitionLocked cache (PLcache) to dynamically lock cache lines. In their design, extra attributes, such as ID, are added to each cache line to indicate the owner. Evictions from different owners are restricted. Similarly, stealthy memory [21, 38] is proposed to lock some cache lines to store sensitive data.

Raj et al. [54] demonstrated how to use cache partition to defend side-channel attacks in VMs based on page coloring. When more VMs are running at the same time, their performance overhead is large since cache is exclusively partitioned using colors. Godfrey and Zulkernine [23] implemented and evaluated selective cache flushing and cache coloring-based cache partitioning in Xen. Experimental results show that selective cache flushing is effective to defend sequential side channels with 15% overhead when testing Apache benchmark. Cache coloring-based cache partition is effective for parallel side channels and the overhead is dependent on the number of partitions and can go up to 30%.

9.4.2 Randomization

Similar to the usage in time fuzzing, randomization is used to obfuscate cache access patterns. Wang and Lee [64] proposed Random Permutation Cache (RPCache) to randomize cache mappings, where the index scheme is decided by the permutation table of each process. They suggested using a Permutation Register Set to avoid looking up permutation table during each access. Although RPCache can thwart most cache-based side-channel attacks with low overhead (1% in SPEC2000 benchmark), there are no fabricated hardware using RPCache yet. Later, Kong et al. [40] extend RPCache using informing loads to allow the critical process to gain control once a miss happens when accessing sensitive data.

In addition to cache level randomization, Pax Project introduced randomization to memory level, named address space layout randomization (ASLR) [58]. Most of modern operating systems, including Linux, Windows, Mac OS, iOS, and Android, have integrated it to defend attacks. The main idea behind this technique is to put address space targets in unpredictable locations to make it harder for the attacker to exploit the desired address. Similar address obfuscation researches are also studied heavily in academia [9, 67]. Recently, Hund et al. [29] proposed a generic side-channel attack to infer the precise location of privileged kernel module, although both user and kernel space are protected by ASLR.

10 Summary

Security is a major concern in personal computers as well as embedded and cyber-physical systems. The cache and memory systems are designed to improve the average performance in these systems. However, the improvement in performance also introduced different kinds of security vulnerabilities in the system. Side-channel attacks are a technique that can break the security protection by exploiting non-functional behaviors. Substantial research efforts have been devoted to this area. This paper surveyed the recent memory-level side-channel attacks and countermeasures, mainly focusing on the timing attacks against cloud and embedded systems. In addition, the encryption implementation holes exploited by these attacks are detailed to provide an insight for improving the security strength of future systems.

References

1. Acıiçmez O (2007) Yet another microarchitectural attack:: exploiting i-cache. In: Proceedings of the 2007 ACM workshop on computer security architecture. ACM, New York CSAW '07, pp 11–18. https://doi.org/10.1145/1314466.1314469
2. Acıiçmez O, Koç ÇK (2009) Microarchitectural attacks and countermeasures. Springer, Boston, pp 475–504. https://doi.org/10.1007/978-0-387-71817-0_18
3. Acıiçmez O, Schindler W, Koç ÇK (2005) Improving Brumley and Boneh timing attack on unprotected SSL implementations. In: Proceedings of the 12th ACM conference on computer and communications security, ACM, New York, CCS '05, pp 139–146. https://doi.org/10.1145/1102120.1102140
4. Acıiçmez O, Schindler W, Koç ÇK (2006) Cache based remote timing attack on the AES. Springer, Berlin, pp 271–286. https://doi.org/10.1007/11967668_18
5. Acıiçmez O, Koç ÇK, Seifert JP (2007) On the power of simple branch prediction analysis. In: Proceedings of the 2Nd ACM symposium on information, computer and communications security, ACM, New York, ASIACCS '07, pp 312–320. https://doi.org/10.1145/1229285.1266999
6. Bates A, Mood B, Pletcher J, Pruse H, Valafar M, Butler K (2012) Detecting co-residency with active traffic analysis techniques. In: Proceedings of the 2012 ACM workshop on cloud computing security workshop, ACM, New York, CCSW '12, pp 1–12. https://doi.org/10.1145/2381913.2381915
7. Bernstein DJ (2005) Cache-timing attacks on AES. Preprint available at http://cr.yp.to/papers.html#cachetiming
8. Bernstein DJ, Lange T, Schwabe P (2012) The security impact of a new cryptographic library. Springer, Berlin, pp 159–176. https://doi.org/10.1007/978-3-642-33481-8_9
9. Bhatkar S, DuVarney DC, Sekar R (2003) Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: USENIX security symposium
10. Biham E (1997) A fast new DES implementation in software. Springer, Berlin, pp 260–272. https://doi.org/10.1007/BFb0052352
11. Bogdanov A, Eisenbarth T, Paar C, Wienecke M (2010) Differential cache-collision timing attacks on AES with applications to embedded CPUs. Springer, Berlin, pp 235–251. https://doi.org/10.1007/978-3-642-11925-5_17
12. Brickell E (2011) Technologies to improve platform security. CHES'11 Invited Talk, Sep 2011, https://www.iacr.org/workshops/ches/ches2011/ presentations/Invited%201/CHES2011_Invited_1.pdf
13. Brickell E, Graunke G, Neve M, Seifert J (2006) Software mitigations to hedge AES, against cache-based software side channel vulnerabilities. IACR Cryptology ePrint Archive 2006:52
14. Brumley D, Boneh D (2003) Remote timing attacks are practical. In: Proceedings of the 12th conference on USENIX security symposium, vol 12. USENIX Association, Berkeley SSYM'03, pp 1–1
15. Chiappetta M, Savas E, Yilmaz C (2015) Real time detection of cache-based side-channel attacks using hardware performance counters. IACR Cryptology ePrint Archive 2015:1034
16. Cleemput JV, Coppens B, De Sutter B (2012) Compiler mitigations for time attacks on modern x86 processors. ACM Trans Archit Code Optim 8(4):23:1–23:20. https://doi.org/10.1145/2086696.2086702
17. Cock D, Ge Q, Murray T, Heiser G (2014) The last mile: an empirical study of timing channels on sel4. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, ACM, New York, CCS '14, pp 570–581. https://doi.org/10.1145/2660267.2660294
18. Coppens B, Verbauwhede I, Bosschere KD, Sutter BD (2009) Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: 2009 30th IEEE symposium on security and privacy, pp 45–60. https://doi.org/10.1109/SP.2009.19
19. Crane S, Homescu A, Brunthaler S, Larsen P, Franz M (2015) Thwarting cache side-channel attacks through dynamic software

diversity. In: 22Nd annual network and distributed system security symposium, NDSS 2015, San diego
20. Daemen J, Rijmen V (1999) AES proposal: Rijndael. version 2, AES submission document, http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf
21. Erlingsson Ú, Abadi M (2007) Operating system protection against side-channel attacks that exploit memory latency. Tech. rep., https://www.microsoft.com/en-us/research/publication/operating-system-protection-against-side-channel-attacks-that-exploit-memory-latency/
22. Ge Q, Yarom Y, Cock D, Heiser G (2016) A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. IACR, Cryptology ePrint Archive 2016:613
23. Godfrey M, Zulkernine M (2014) Preventing cache-based side-channel attacks in a cloud environment. IEEE Transactions on Cloud Computing 2(4):395–408. https://doi.org/10.1109/TCC.2014.2358236
24. Gruss D, Spreitzer R, Mangard S (2015) Cache template attacks: automating attacks on inclusive last-level caches. In: 24Th USENIX security symposium (USENIX security 15). USENIX Association, Washington, D.C., pp 897-912
25. Gruss D, Maurice C, Wagner K, Mangard S (2016) Flush+flush: a fast and stealthy cache attack. In: Proceedings of the 13th international conference on detection of intrusions and malware, and vulnerability assessment, vol 9721, Springer, New York, Inc., DIMVA 2016, pp 279–299. https://doi.org/10.1007/978-3-319-40667-1_14
26. Gullasch D, Bangerter E, Krenn S (2011) Cache games—bringing access-based cache attacks on AES to practice. In: Proceedings of the 2011 IEEE symposium on security and privacy, IEEE Computer Society, Washington, SP '11, pp 490–505. https://doi.org/10.1109/SP.2011.22
27. Gülmezoğlu B, İnci MS, Irazoqui G, Eisenbarth T, Sunar B (2015) A faster and more realistic flush+reload attack on AES. Springer International Publishing, Cham, pp 111–126. https://doi.org/10.1007/978-3-319-21476-4_8
28. Hamburg M (2009) Accelerating AES with vector permute instructions. Springer, Berlin, pp 18–32
29. Hund R, Willems C, Holz T (2013) Practical timing side channel attacks against kernel space ASLR. In: 2013 IEEE symposium on security and privacy, pp 191–205. https://doi.org/10.1109/SP.2013.23
30. İnci MS, Gülmezoğlu B, Eisenbarth T, Sunar B (2016) Co-location detection on the cloud. Springer International Publishing, Cham, pp 19–34. https://doi.org/10.1007/978-3-319-43283-0_2
31. İnci MS, Gülmezoğlu B, Irazoqui G, Eisenbarth T, Sunar B (2016) Cache attacks enable bulk key recovery on the cloud. In: Cryptographic hardware and embedded systems - CHES 2016 - 18th international conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings, pp 368–388. https://doi.org/10.1007/978-3-662-53140-2_18
32. Irazoqui G, Inci MS, Eisenbarth T, Sunar B (2014) Fine grain cross-vm attacks on xen and vmware. In: Proceedings of the 2014 IEEE fourth international conference on big data and cloud computing, IEEE Computer Society, Washington, BDCLOUD '14, pp 737–744. https://doi.org/10.1109/BDCloud.2014.102
33. Irazoqui G, Inci MS, Eisenbarth T, Sunar B (2014) Wait a minute! A fast, cross-VM attack on AES. Springer International Publishing, Cham, pp 299–319. https://doi.org/10.1007/978-3-319-11379-1_15
34. Irazoqui G, Eisenbarth T, Sunar B (2015) S$A: a shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In: 2015 IEEE symposium on security and privacy, pp 591–604. https://doi.org/10.1109/SP.2015.42
35. Käsper E, Schwabe P (2009) Faster and timing-attack resistant AES-GCM. Springer, Berlin, pp 1–17. https://doi.org/10.1007/978-3-642-04138-9_1
36. Kelsey J, Schneier B, Wagner D, Hall C (2000) Side channel cryptanalysis of product ciphers. J Comput Secur 8(2,3):141–158
37. kernelorg (2009) Address space layout randomization (ASLR). https://www.kernel.org/doc/Documentation/vm/ksm.txt
38. Kim T, Peinado M, Mainar-Ruiz G (2012) Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In: Presented as part of the 21st USENIX security symposium (USENIX security 12). Bellevue, USENIX, pp 189-204
39. Kocher PC (1996) Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. Springer, Berlin, pp 104–113. https://doi.org/10.1007/3-540-68697-5_9
40. Kong J, Aciicmez O, Seifert JP, Zhou H (2009) Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: 2009 IEEE 15th international symposium on high performance computer architecture, pp 393–404. https://doi.org/10.1109/HPCA.2009.4798277
41. Lipp M (2016) Cache attacks on arm. Master thesis, Graz, University Of Technology
42. Lipp M, Gruss D, Spreitzer R, Maurice C, Mangard S (2016) Armageddon: cache attacks on mobile devices. In: 25Th USENIX security symposium (USENIX security 16). USENIX association, Austin, pp 549–564
43. Liu F, Yarom Y, Ge Q, Heiser G, Lee RB (2015) Last-level cache side-channel attacks are practical. In: 2015 IEEE symposium on security and privacy, pp 605–622. https://doi.org/10.1109/SP.2015.43
44. Martin R, Demme J, Sethumadhavan S (2012) Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. SIGARCH Comput Archit News 40(3):118–129. https://doi.org/10.1145/2366231.2337173
45. Matsui M, Nakajima J (2007) On the power of bitslice implementation on Intel core2 processor. Springer, Berlin, pp 121–134. https://doi.org/10.1007/978-3-540-74735-2_9
46. Montgomery PL (1985) Modular multiplication without trial division. Math Comput 44:519–521
47. Oren Y, Kemerlis VP, Sethumadhavan S, Keromytis AD (2015) The spy in the sandbox: practical cache attacks in javascript and their implications. In: Proceedings of the 22Nd ACM SIGSAC conference on computer and communications security, ACM, New York, CCS '15, pp 1406–1418. https://doi.org/10.1145/2810103.2813708
48. Osvik DA, Shamir A, Tromer E (2006) Cache attacks and countermeasures: the case of AES. Springer, Berlin, pp 1–20. https://doi.org/10.1007/11605805_1
49. Owens R, Wang W (2011) Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In: 30th IEEE international performance computing and communications conference, pp 1–8. https://doi.org/10.1109/PCCC.2011.6108094
50. Page D (2002) Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, http://eprint.iacr.org/2002/169
51. Page D (2005) Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280, http://eprint.iacr.org/2005/280
52. Payer M (2016) HexPADS: a platform to detect "Stealth" attacks. Springer International Publishing, Cham, pp 138–154. https://doi.org/10.1007/978-3-319-30806-7_9
53. Percival C (2005) Cache missing for fun and profit. BSDCan 2005

54. Raj H, Nathuji R, Singh A, England P (2009) Resource management for isolation enhanced cloud services. In: Proceedings of the 2009 ACM workshop on cloud computing security, ACM, New York, CCSW '09, pp 77–84. https://doi.org/10.1145/1655008.1655019
55. Ristenpart T, Tromer E, Shacham H, Savage S (2009) Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM conference on computer and communications security, ACM, New York, CCS '09, pp 199–212. https://doi.org/10.1145/1653662.1653687
56. Spreitzer R, Gérard B (2014) Towards more practical time-driven cache attacks. Springer, Berlin, pp 24–39. https://doi.org/10.1007/978-3-662-43826-8_3
57. Spreitzer R, Plos T (2013) On the applicability of time-driven cache attacks on mobile devices. Springer, Berlin, pp 656–662. https://doi.org/10.1007/978-3-642-38631-2_53
58. Team P (2003) Address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt
59. Tromer E, Osvik DA, Shamir A (2010) Efficient cache attacks on AES, and countermeasures. J Cryptol 23(1):37–71. https://doi.org/10.1007/s00145-009-9049-y
60. Tsunoo Y, Saito T, Suzaki T, Shigeri M, Miyauchi H (2003) Cryptanalysis of DES implemented on computers with cache. Springer, Berlin, pp 62–76. https://doi.org/10.1007/978-3-540-45238-6_6
61. Varadarajan V, Ristenpart T, Swift M (2014) Scheduler-based defenses against cross-vm side-channels. In: 23Rd USENIX security symposium (USENIX security 14). USENIX Association, San Diego, pp 687–702
62. Varadarajan V, Zhang Y, Ristenpart T, Swift M (2015) A placement vulnerability study in multi-tenant public clouds. In: 24Th USENIX security symposium (USENIX security 15). USENIX Association, Washington, pp 913–928
63. Vattikonda BC, Das S, Shacham H (2011) Eliminating fine grained timers in xen. In: Proceedings of the 3rd ACM workshop on cloud computing security workshop, ACM, New York, CCSW '11, pp 41–46. https://doi.org/10.1145/2046660.2046671
64. Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. SIGARCH Comput Archit News 35(2):494–505. https://doi.org/10.1145/1273440.1250723
65. Weiß M, Heinz B, Stumpf F (2012) A cache timing attack on AES in virtualization environments. Springer, Berlin, pp 314–328. https://doi.org/10.1007/978-3-642-32946-3_23
66. Wu Z, Xu Z, Wang H (2012) Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In: Presented as part of the 21st USENIX security symposium (USENIX security 12). Bellevue, USENIX, pp 159–173
67. Xu J, Kalbarczyk Z, Iyer RK (2003) Transparent runtime randomization for security. In: 22nd international symposium on reliable distributed systems, 2003. Proceedings, pp 260–269. https://doi.org/10.1109/RELDIS.2003.1238076
68. Xu Z, Wang H, Wu Z (2015) A measurement study on co-residence threat inside the cloud. In: 24Th USENIX security symposium (USENIX security 15). USENIX Association, Washington, pp 929–944
69. Yarom Y, Benger N (2014) Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive 2014:140
70. Yarom Y, Falkner K (2014) FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: 23Rd USENIX security symposium (USENIX security 14). USENIX association, San Diego, pp 719–732
71. Yarom Y, Genkin D, Heninger N (2017) Cachebleed: a timing attack on OpenSSL constant-time RSA. J Cryptogr Eng :1–14. https://doi.org/10.1007/s13389-017-0152-y
72. Zhang X, Xiao Y, Zhang Y (2016) Return-oriented flush-reload side channels on arm and their implications for android devices. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM, New York, CCS '16, pp 858–870. https://doi.org/10.1145/2976749.2978360
73. Zhang Y, Reiter MK (2013) Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: 20th ACM SIGSAC conference on computer and communications security. ACM, New York, pp 827–838
74. Zhang Y, Juels A, Oprea A, Reiter MK (2011) Homealone: co-residency detection in the cloud via side-channel analysis. In: 2011 IEEE symposium on security and privacy, pp 313–328. https://doi.org/10.1109/SP.2011.31
75. Zhang Y, Juels A, Reiter MK, Ristenpart T (2012) Cross-VM side channels and their use to extract private keys. In: Proceedings of the 2012 ACM conference on computer and communications security. ACM, New York, CCS '12, pp 305–316. https://doi.org/10.1145/2382196.2382230
76. Zhang Y, Juels A, Reiter MK, Ristenpart T (2014) Cross-tenant side-channel attacks in PaaS clouds. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. ACM, New York, CCS '14, pp 990–1003. https://doi.org/10.1145/2660267.2660356
77. Zhou Z, Reiter MK, Zhang Y (2016) A software approach to defeating side channels in last-level caches. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM, New York, CCS '16, pp 871–882,. https://doi.org/10.1145/2976749.2978324