Chapter 18
Microarchitectural Attacks and Countermeasures

Onur Acıiçmez and Çetin Kaya Koç

18.1 Introduction

Microarchitectural analysis (MA) is a fast evolving area of side-channel cryptanalysis. This new area focuses on the effects of common processor components and their functionalities on the security of software cryptosystems. The main characteristic of microarchitectural attacks, which sets them aside from classical side-channel attacks, is the simple fact that they exploit the microarchitectural behavior of modern computer systems.

The fascinating progress of microprocessor technology in the last decades is maybe the most influential power that has been driving the scientific and technological advances. However, due to strictly throughput, performance, and "performance per watt"-oriented goals of modern processor designs and also "time-to-market"-driven business philosophy, the resulting products, i.e., commodity processor architectures in the market, lack a thorough security analysis. The main element that gave birth to microarchitectural analysis area is indeed this particular gap between the current processor architectures and the ideal secure computing environment.

The identification of requirements for secure execution environments has always been a challenging task since the invention of high-complexity computing devices. The security requirements of early computer systems were defined with monolithic mainframe computers in mind (cf. [9, 15, 48] and also [19] for a nice collection of early computer security efforts). Today, the domination of multi-user PC and server platforms and also the multitasking operating systems mandates a serious revision of these early requirements. Recently, we have seen an increased effort on the security analysis of daily life computer platforms. The advances in the field, more specifically, the desire to develop secure execution technologies such as AMD's Pacifica, Intel's virtualization technology (VT) and trusted execution technology (TXT) (codenamed LaGrande technology or LT for short) play an important role to increase

Samsung Information Systems America, e-mail: onur.aciicmez@gmail.com · City University of Istanbul & University of California Santa Barbara, e-mail: koc@cryptocode.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_18,

attention on analysis of computer platform security due to [42]. Here, it has been especially shown that microarchitectural properties of modern processors create a significant security risk (cf. [3, 4, 6, 10, 29, 35]).

Today's high-end computer architectures employ several different components each of which is responsible for a specific task mostly to increase the performance of the system. Among all these different components, we will focus on only four of them in this chapter:

1. Data cache
2. Branch prediction unit
3. Instruction cache
4. Functional units, especially multiplier

These four components are the ones that had been exploited in MA until the time this chapter was written. Although it is necessary to understand the detailed functionality and purposes of these components in order to grasp the basic idea underlying the theory of MA, we cannot cover all these details in this chapter. It would take yet another book to explain even the basics of modern computer architecture, and therefore we have to assume that the reader already has at least some familiarity with computer architecture concepts. There are several books in the literature (e.g., [16, 34, 38, 39]) that give comprehensive overviews of modern computer architectures. Even though we will try to give very brief explanations of the aforementioned microprocessor components, we recommend the readers to study the related materials from [34, 39] or a similar resource in advance.

In this chapter, we cover all of these four MA types mentioned above. We start with an overview and history of microarchitectural analysis. Then we present each MA type including the basics of these attacks and examples of concrete attack strategies found in the literature. We also discuss the differences between these MA types and possible countermeasure techniques.

18.2 Overview and Brief History

The actual origins of microarchitectural analysis go back to [20, 40]. Although these publications implicitly pointed out the security risks of microprocessor components like cache, concrete and widely applicable security attacks relying on microprocessor functionalities have not been worked out until very recent years. The results of these recent studies immediately attracted significant public interest due to their implications and broad application ranges of these security breaches.

The typical targets of side-channel analysis have been and still are smart cards. However, we have seen significant increase in the research efforts spent on side-channel analysis of commodity PC platforms. Soon, the researchers realized the fact that the internal functionalities of some microprocessor components like data and instruction cache and branch prediction units cause very serious side-channel leakage and hence create crucial security risks. These efforts led to the development of microarchitectural analysis area.

Side-channel analysis can be defined as the study of the relations between the strength of cryptosystems and data-dependent variations in the so-called side-channel information, e.g., execution time and power consumption, generated during the execution of their physical implementations. Malicious parties can exploit such variations to find out the secrets used in security applications and cryptosystems. These variations either directly give the key value out during a single cipher execution or leak sensitive information which can be gathered during many executions and analyzed to compromise the system. MA attacks exploit the microarchitectural components of a processor to reveal cryptographic keys. The internal functionalities of the aforementioned processor components generate such data-dependent variations in execution time and power consumption, which are the subjects of MA.

The first type of MA we had seen is called "Cache Analysis". A cache-based attack, abbreviated to "cache attack" or "cache analysis" from here on, exploits the cache behavior of a cryptosystem by obtaining the execution time and/or power consumption variations generated via cache hits and misses, cf. [5, 6, 10–12, 24, 26, 27, 29, 31, 35, 43–45]. The potential cache vulnerability of computer systems has been known for a long time, cf. [20, 22, 23]; however, actual realistic and practical cache attacks were not developed until recent years. Cache analysis techniques enable an unprivileged process to attack another process, e.g., a cipher process, running in parallel on the same processor as done in [26, 29, 35]. Furthermore, some of the cache attacks can even be carried out remotely, i.e., over a network [6].

The current cache attacks in the literature, excluding instruction cache attacks which are fundamentally different than data cache attacks, are data-path attacks. They exploit the data access patterns of a cipher. The memory accesses of S-box-based ciphers like DES and AES are key dependent. Cache attacks analyze the cache statistics, e.g., miss/hit rates, of the cipher execution and try to reveal these memory access patterns. Cache statistics of an execution include the number of cache hits and misses, the cache lines modified by the cipher, and such. An unprivileged malicious party cannot directly obtain the cache statistics of a cipher,[1] but it can observe the side-channel leakage through execution time and/or power consumption to estimate these values. For instance, the execution time of AES software implementations is directly related to the total number of cache hits and misses occuring during an encryption, cf. [41], and someone can measure AES encryption time to determine these statistics.

Branch prediction analysis (BPA) is the second type of MA which was developed in 2006. Several variants of BPA attacks were introduced in [4], all of which exploit the side-channel leakage due to branch prediction units of microprocessors. The most powerful variant of BPA is called simple branch prediction analysis (SBPA) and it relies on the ability to run a spy process parallel to the cipher process under

[1] In fact, current processors have special registers, called performance counters, inside the chip to count and store such statistics. These registers are mainly used for performance monitoring purposes and fortunately require special privileges to be read. The potential power of a malicious party would be significantly higher without this requirement of high privilege. For further information on performance counters and performance monitoring events, refer to [49].

attack [3]. According to [3], a carefully written spy process running simultaneously with an RSA process is able to collect during one *single* RSA signing execution almost all of the secret key bits. The concept of SBPA is proved in [3] by applying an attack on the exponentiation phase of a simple RSA implementation as a case study. A spy process, which relies on the simultaneous multithreading (SMT) capability of some microprocessors, is implemented to observe the execution of an RSA cipher process. This concept was also verified by André Seznec, a well-known expert on branch prediction [50].

The actual power of SBPA is not limited to this basic application on RSA exponentiation. The SBPA has a potential to reveal the entire execution flow of a target process on *almost any* execution environment, i.e., with or without SMT. This is a very strong claim which has not been experimentally verified.

Following this interesting research field, two other MA are also introduced: exploiting instruction cache (I-cache analysis) and shared functional units (SFU analysis). Similar to BPA, I-cache and SFU analysis rely on spy routines and they reveal the execution flow of cryptosystems. In I-cache analysis, an adversary runs a spy process simultaneously or quasi-parallel with the cipher and detects the changes occuring in the instruction cache.

The principles of SFU analysis are different than the previous MA types. The previous types, i.e., cache, branch prediction, and instruction cache analysis, try to observe the changes in the *persistent state* of the mentioned microprocessor components. The spy process-oriented MA attacks, except SFU analysis, rely on the fact that the execution of cryptosystems leaves *persistent* changes in the state of shared resources like cache and branch target buffer. In other words, the cipher execution leaves "footprints" on the observable state, i.e., the so-called metadata of these resources and an unprivileged spy process can keep track of these footprints if it runs on the same processor in parallel with the cipher. An adversary can reveal the execution flow and/or the memory access patterns of cryptosystems by spying on these states and especially by detecting the changes of these states as a function of time. On the other hand, SFU analysis does not take advantage of persistent states. It follows a quite different approach and tries to detect when a certain functional unit is occupied by the cipher.

We will explain the details of each of these MA types in the following sections.

18.3 Cache Analysis

18.3.1 Basics of Cache

We can only give a brief explanation of cache in this section. We recommend the readers to explore more on cache architectures in order to grasp the details of cache attacks. For further information on cache, please refer to [17, 18, 30].

A high-frequency processor needs to retrieve the data at a very high speed in order to utilize its functional resources. The latency of a main memory is not

short enough to match this demand of high-speed data delivery. The gap between the latency of main memories and the actual demand of processors has been and will be continuously increasing as Moore's law holds. Common to all processors, the attempt to close this gap is the employment of a special buffer called cache.

A cache is a small and fast storage area used by a CPU to reduce the average memory access time. It acts as a buffer between the main memory and the processor core and provides the processor fast and easy access to the most frequently used data (including instructions) without frequent external bus accesses.

Cache stores the copies of the most frequently used data. When the processor needs to read a location in main memory, it first checks to see if the data are already in the cache. If the data are already in the cache (called a cache hit), the processor immediately uses this data instead of accessing the main memory, which has a significantly longer latency than a cache. Otherwise (a cache miss), the data are read from the memory and a copy of it is stored in the cache. This copy is expected to be used in the near future due to the temporal locality property.

A cache is partitioned into a number of non-overlapping fixed size blocks, called cache blocks or cache lines. The minimum amount of data that can be read from the main memory into a cache at once is called cache line or cache block size, i.e., each cache miss causes a cache block to be retrieved from a higher level memory. The reason why a block of data is transferred from the main memory to the cache instead of transferring only the data that are currently needed lies in spatial locality property. Since a cache is limited in size, storing new data in a cache mandates eviction of some of the previously stored data.

The method of deciding where to store and search for a data in a cache is called cache mapping strategy. Three main cache mapping strategies are direct, fully associative, and set associative mapping.

A particular data block can only be stored in a single certain location in a direct mapped cache. The exact location is determined using the address of the data block. On the contrary, a data block can be placed in potentially any location in a fully associative cache. The location of a particular placement is determined by the replacement policy. Set associative mapping is a combination of these two mapping strategies. Set associative caches are divided into a number of same size sets, called cache sets, and each set contains the same fixed number of cache lines. A data block can be stored only in a certain cache set based on the address of the data block (just like in a direct mapped cache); however, it can be placed in any location inside this set (like in a fully associative cache). Again, the particular location of a data inside its cache set is determined by the replacement policy.

The replacement policy is the method of deciding which data block to evict from the cache in order to place the new one in. The ultimate goal is to choose the data that are most unlikely to be used in the near future. There are several cache replacement policies proposed in the literature (cf. [18, 34]). In this document, we focus on a specific one: least recently used (LRU). It is the most commonly used policy and it picks the data that are least recently used among all of the candidate data blocks that can be evicted from the cache.

18.3.2 Overview of Cache Attacks

Cryptosystems have data-dependent memory access patterns. For example, S-box-based block ciphers like DES and AES employ table lookups and the indices of these lookups are functions of the plaintext and the secret key. Cache architectures leak information about the cache hit/miss statistics of ciphers through side-channels, e.g., execution time and power consumption. Therefore, it is possible to exploit cache behavior of a cipher to obtain information about its memory access patterns, i.e., indices of S-box and table lookups.

Cache attacks rely on cache hits and misses occur during the encryption/decryption process of a cryptosystem. Even if a cipher implementation has a fixed execution flow, i.e., if the same instructions are executed for any particular (plaintext, cipherkey) pair, the cache behavior during the execution causes variations in the program execution time. Cache attacks exploit such variations and narrow the exhaustive search space of secret keys.

Theoretical cache attacks were first described by Page in [31]. He characterized two types of cache attacks: trace driven and time driven. We have recently seen another type of cache attacks that can be named as "access-driven" attacks.

In trace-driven cache attacks, the adversary obtains the traces of cache activity for a sample of encryptions. We define a trace as a sequence of cache hits and misses. For example,

$$MHHMHMHM, MMHMHHMH, MMMMHHHH$$

are examples of a trace of length 8. Here H and M represent a cache hit and miss, respectively. The first memory access in the first example results in a miss, second one in a hit, and so on. If an adversary captures such traces, he can determine whether a particular access during an encryption is a hit or miss. Therefore, the adversary has the ability to observe (e.g.) if the second access to a lookup table yields a hit and can infer information about the lookup indices, which are key dependent. This ability gives an adversary the opportunity to make inferences about the secret key.

Time-driven attacks, on the other hand, are less restrictive and they do not rely on the ability of capturing the outcomes of individual memory accesses. Adversary observes the aggregate profile, i.e., total number of cache hits and misses or at least a value that can be used to approximate these numbers. For example, he measures the total execution time of a cipher and uses this measurement to approximate the number of cache misses occurring during the encryption. Note that each cache miss introduces a delay to the overall execution time and thus the total encryption time is proportional to the number of cache misses. Time-driven attacks are based on statistical inferences and therefore require much higher number of samples than trace-driven attacks.

While trace-driven and time-driven attacks analyze the outcomes of memory accesses, access-driven attacks follow a different approach. The adversary determines the cache sets that the cipher process modifies. Therefore, he can understand which

elements of the lookup tables or S-boxes are accessed by the cipher. Then, the candidate keys that cause an access to unaccessed parts of the tables can be eliminated.

In the following sections, we explain each of these cache attack types in more detail. We describe simplified attack models for each type and try to enrich the understanding of the reader by showing concrete attack examples from the literature along with these models. We can only focus on a small fraction of the previous studies on this subject in this chapter to keep the length in a reasonable range. Therefore, we first want to give a short survey on cache analysis and briefly cover the entire prior art before delving into the details of our set of concrete attack examples.

18.3.3 A Brief Survey on Cache Analysis

Although [20, 22, 23] pointed the cache vulnerability of computer systems a long time ago, actual realistic cache attacks had not been developed until recent years. D. Page described and simulated a theoretical cache attack on DES [31] in 2002. Actual cache-based timing attacks were first implemented by Tsunoo et al. [43, 44]. They developed several cache attacks on various ciphers, including MISTY1, DES, and Triple-DES [43]. Their original attack on MISTY1 was improved later in [45].

Bernstein showed the vulnerability of AES software implementations on various platforms [10]. There was a common belief that Bernstein's attack could be used as a real remote attack; however, later studies proved it wrong [27].

Osvik et al. described various local cache attack variants first in [28] in 2005, then they presented their results at CT-RSA in early 2006 [29]. They made use of a local process, called spy process, to monitor the cache activities of an AES process. They exploited the collisions between the table lookups in the first two rounds of AES and the memory access operations of the spy process. Neve et al. improved these attacks in [29] by taking the last AES round into consideration [26].

The same idea of exploiting collisions between two different processes was also used by Percival in [35] and Bertoni et al. in [11]. Percival exploited simultaneous multithreading feature of the modern processors and developed a cache attack on RSA [35]. Bertoni et al. developed a cache-based power attack on AES using the idea of exploiting external collisions.

Similar to external collisions between different processes, one can also exploit internal collisions inside a cipher. The attacks of Tsunoo et al. are based on this principle [43, 44]. Trace-driven attacks also rely on internal collisions [5, 24]. A summary of possible cache collision attacks on AES is given in [12].

None of these efforts was successful to achieve the goal of developing a generic and universally applicable cache attack that can also compromise remote systems. A remote cache attack and ideas to develop a universal remote cache attack on AES are given in [6].

Several hardware- and software-based countermeasures were proposed to prevent cache attacks in [10, 13, 29, 32, 33].

18.3.4 Time-Driven and Trace-Driven Attacks

Let P_i and K_i be the ith byte of the plaintext and cipherkey, respectively. In this chapter, each byte is considered to be either an 8-digit radix-2 number, i.e., $\in \{0,1\}^8$, that can be added in GF(2^8) using a bitwise exclusive OR operation or an integer in [0, 255] that can be used as an index.

Assume that we have a cryptosystem, which operates on a lookup table, and each element of this table is as large as the size of a cache line. There are two accesses to the same lookup table as in Figure 18.1. The indices of these accesses are a function of different bytes of the plaintext and the cipherkey. An adversary removes all of the cipher data from the cache by (e.g.) reading or writing a large piece of data. Therefore, prior to an encryption the cache does not contain any data that belongs to this cryptosystem.

Then the adversary triggers encryption with arbitrary plaintext. Whenever the two indices become equal for a plaintext, the second access will find the target data in the cache and result in a cache hit. In other words, whenever the equation

$$(P_1 \oplus K_1 = P_2 \oplus K_2)$$

holds for a plaintext, or in a different interpretation, the equation

$$(P_1 \oplus P_2 = K_1 \oplus K_2)$$

holds, then we will have a cache hit in the second during the encryption of this plaintext. Note that we assume a clean cache prior to the encryption, i.e., the cache does not contain any data belonging to this cipher. Similarly, whenever there is a cache hit in the second table lookup, these equations will also hold. Therefore, the key byte difference $K_1 \oplus K_2$ can be derived from the values of plaintext bytes P_1 and P_2 if this plaintext causes such a cache hit.

In trace-driven attacks, we assume that the adversary can directly understand if the latter access results a hit, thus he can directly obtain $K_1 \oplus K_2$.

This goal is more complicated in time-driven attacks. These attacks rely on the following facts:

- The execution time of a cipher is directly related to the number of cache misses occurring during the execution. In other words, the overall execution time of an encryption can be used to approximate the number of cache misses that occur during the encryption.

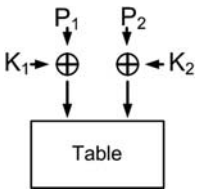

Fig. 18.1 Two different accesses to the same table.

- The average number of cache misses in encryptions of a sample of plaintext that results in a biased cache miss is different than the case of a random sample. For example, a large sample of plaintext, each of which results in a cache miss during the second table lookup, causes a different number of cache misses in average during the execution of a cipher compared to a random sample that does not result in such a biased cache miss.

The expected number of cache misses during an encryption with a plaintext that obeys the equation $(P_1 \oplus P_2 = K_1 \oplus K_2)$ is less than the expected number of cache misses when the plaintext does not obey this equation. We need to use a large sample to realize an accurate statistics of the execution. In our case, if we collect a sample of different plaintext with the corresponding execution time, the plaintext byte difference, $P_1 \oplus P_2$, that causes least number of cache misses in average (i.e., the shortest average execution time) gives the correct key byte difference.

When we consider this fact, a simple attack method becomes the following:

Phase 1: Obtain a sample of (plaintext, encryption time) pairs generated under the same target key.

Phase 2: Split this initial set into 128 subsets based on the plaintext values. In order to do this, first create a subset for each possible value of $P_1 \oplus P_2$. Note that there are 128 possibilities because P_1 and P_2 are bytes and therefore the length of $P_1 \oplus P_2$ is 8 bits. Then scan each plaintext in the initial sample and put it in the subset that corresponds to $P_1 \oplus P_2$ value of this plaintext.

Phase 3: Calculate the average encryption time of the entities in each subset. If the initial sample obtained in Phase 1 is large enough, all of these average values, except one subset, will be close to each other. The only exception is the subset that corresponds to the $P_1 \oplus P_2$ value that is equal to $K_1 \oplus K_2$. Therefore, the key byte difference can be recovered.

In this basic example attack, we assume that each element of the lookup table is as large as the size of a cache line. However, the elements of the lookup tables in real implementations are usually smaller than the cache line size. Therefore, each cache line stores more than one element of the table. Any cache miss results in the transfer of an entire cache line, not only a single element, from the main memory. This indicates the fact that there will be a cache hit even when $P_1 \oplus K_1$ is not equal to $P_2 \oplus K_2$. If a cache line stores more than one element of a table, those will be consecutive elements of the table because of the current cache architectures. Hence, we can still recover the most significant bits of $K_1 \oplus K_2$ in a real attack by following our basic attack model. This concept will be more clear when we cover our first concrete cache attack in the next section.

18.3.5 Exploiting Internal Collisions in Time-Driven Attacks

We cover some example cache attacks on AES-128 in this section. Our first example is very similar to the basic attack given above.

The most widely used AES software implementation employs five different lookup tables. There are 10 rounds of computations in AES-128 and 16 table lookups in each round. The indices of the first round table lookups are in the form $P_i \oplus K_i$ for $i \in \{0, 1, ..., 15\}$. The indices to the first AES table in the first round are $P_0 \oplus K_0$, $P_4 \oplus K_4$, $P_8 \oplus K_8$, and $P_{12} \oplus K_{12}$.

We can directly apply the idea given in the previous section to these indices. If we use the simple attack method on the first two indices $P_0 \oplus K_0$ and $P_4 \oplus K_4$ by splitting our initial sample set into the subsets based on the value $P_0 \oplus P_4$, the distinct average encryption time values will indicate the value of $K_0 \oplus K_4$.

Figure 18.2 shows the average execution time for each subset that was formed during the search of $K_0 \oplus K_4$ in a real attack using the indices $P_0 \oplus K_0$ and $P_4 \oplus K_4$. It can clearly be seen in this figure that there are several distinct points as opposed to a single point for the correct $K_0 \oplus K_4$ value. The reason, as explained above, is that the elements of the lookup tables in real implementations are usually smaller than the actual cache line size and thus each cache line stores more than one element of

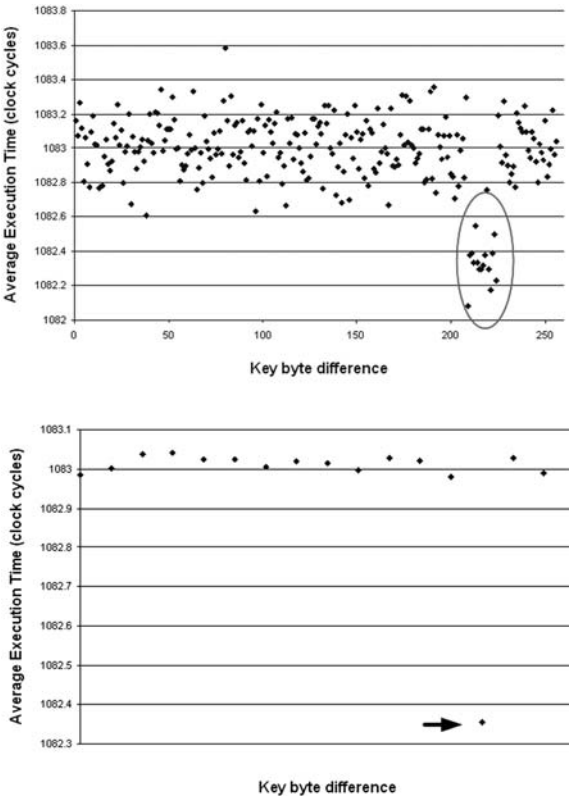

Fig. 18.2 The first graph shows the results of the search for $\langle K_0 \oplus K_8 \rangle_{msp}$ using the indices $(P_0 \oplus K_0)$ and $(P_8 \oplus K_8)$ in an experimental attack. The second graph shows the same search when the number of table elements in a single cache line is known to be 16.

the table. Therefore, there is a cache hit during the latter table lookup whenever the most significant parts of $P_0 \oplus K_0$ and $P_4 \oplus K_4$ become equal for a plaintext. So, we can only find the difference of the most significant part of the key bytes using the equation

$$\langle P_0 \rangle \oplus \langle P_4 \rangle = \langle K_0 \rangle \oplus \langle K_4 \rangle$$

where $\langle A \rangle$ stands for the most significant part of A. The size of the most significant parts, i.e., the number of bits that can be recovered by exploiting these two indices, depends on how many table elements a cache line holds.

The dashed ellipse in the top graph of this figure obviously contains more than 8 and less than 32 points, so an attacker can conclude that there are 16 table elements in a cache line, which also means that the most significant parts of the key byte differences are 4 bits long. Using less sets during a search on key byte differences gives more clear identification of the correct value, because each set contains more elements in this case. An increase in the size of a sample gives a better estimation of the expected execution time for this sample because the variance of the average encryption time decreases proportional to the size of the sample. The bottom graph shows the average execution time of each 16 sets that can be formed for the same search.

Applying the same idea on different indices of the first round, we can find the key byte differences $\langle K_i \oplus K_{4*j+i} \rangle$ with $i, j \in \{0,1,2,3\}$. This attack is called first round attack. On current widely used processors the search space can typically be reduced to 56, 68, or 80 bits for 128-bit keys. If we also consider the indices of the second round table lookups, called second round attack or two-round attack, we can reduce the search space to 32 bits. The equations used in second round attack are more complicated than first round attack; therefore, we do not cover them in detail in this chapter. Further details can be found in [6].

These attack principles can be used to develop a remote cache attack, i.e., a cache attack that does not require local access to the target machine and can compromise the systems over a network just by sending messages to the systems and measuring the response time. They also showed how one can devise and apply such remote cache attacks on other cryptosystems. Their experiments indicate that cache attacks can be used to extract secret keys of remote systems if the system under attack runs on a server with a multitasking or multithreading system and a large enough workload. Although a large number of measurements are required to successfully perform a remote cache attack, it is feasible in principle.

18.3.6 Access-Driven Attacks

In a computer system, we have a main memory, which stores the data of each process running on the system, and a cache between the memory and the CPU as shown in Figure 18.3. We represent each cache block with a square in this figure and each column corresponds to a different cache set. For example, the cache in this figure has two blocks in each column, so it is two-way set associative. Assume that the data

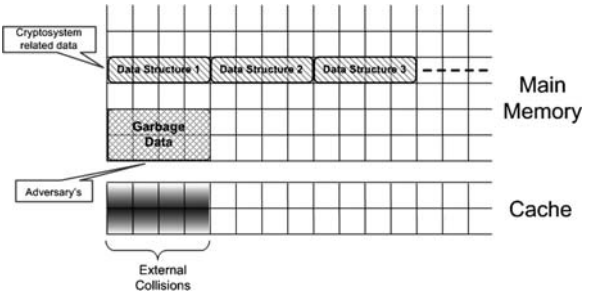

Fig. 18.3 Main memory and cache structure in a CPU.

blocks in a column of the memory map only to the corresponding cache set in the same column of the cache. Mapping a memory block to a cache set means that this particular cache block can only be stored in that set of the cache. As an example, the garbage data and data structure 1 can only be in the dark area of the cache in Figure 18.3. In fact, the mapping between memory locations and cache blocks in real computer systems are different and more complicated than our basic assumption. However, for the sake of simplicity and clarity, we follow this assumption in this chapter.

Also assume the following. There are two different processes, the cryptosystem and a malicious process, called Spy process, running on the same machine. Cryptosystem process operates on several different data structures. The actual value of the secret key affects which of these data structures (e.g., which parts of a table lookup) are accessed during an encryption and when (e.g., in which round) they are accessed.

An adversary can easily understand if the cipher has at least one access to, for example, data structure 1 during a particular encryption. This is due to the situation that accesses to garbage data and data structure 1 create external collisions. We define a collision in this context as the situation that occurs when an attempt is made to store two or more different data items at a location that can hold only one of them. We use the term "external collision" if these data items belong to different processes. On the other hand, if the data items belong to the same process, we call it as "internal collision".

In our case here, the cache does not have enough number of sets to store the garbage data and data structure 1 at the same time. Since the cache is only two-way associative and the garbage data completely fills the dark area in, an access to garbage data results in replacing any previously stored data in the dark area. Similarly, an access to any data that maps to the same cache location also replaces some or all of the garbage data if it resides in the cache prior to this access. Therefore, an access to the garbage data may evict data structure 1 from the cache and vice versa. This fact enables an adversary to devise an attack on the cryptosystem process.

A basic attack works as the following. An adversary reads the garbage data which would force the CPU to load the content of it into the cache (Figure 18.4a). Then

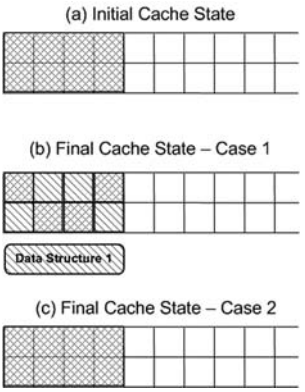

Fig. 18.4 Steps of a basic attack.

the adversary triggers an encryption and the cryptosystem is run under this initial cache state. There are two possible cases that may happen during the encryption:

Case 1: Cipher accesses data structure 1.
Case 2: Cipher does not access data structure 1.

When the first case happens, the access to data structure 1 changes the content of the first four cache sets as shown in the figure. Otherwise, these sets remain unchanged. In Case 1 (Case 2, resp.) the final state of the first four cache sets just after the encryption becomes like Figure 18.4b (Figure 18.4c, resp.). When the adversary reads the garbage data again after the encryption, he can understand which case was true, because reading the garbage data creates some cache misses and thus takes longer in Case 1. Similarly, at least in theory, the adversary can use the same technique for other data structures and reveal the entire set of items that are accessed during an encryption. Since this set depends on the secret key value, he can gain invaluable information to narrow the exhaustive search space.

This model describes an active attack where the adversary must be able to control the contents of the cache. The cache attacks that rely on this basic model correspond to access-driven types. Percival's attack on RSA [35] (cf. Section 18.3.7), Osvik et al. and Neve et al.'s attacks on AES [26, 28, 29] (cf. Section 18.3.6.1), and the power attack by Bertoni et al. [11] use this attack model. In fact, branch prediction analysis (cf. Section 18.4) and instruction cache analysis (cf. Section 18.5) are also based on similar approaches.

18.3.6.1 Osvik–Shamir–Tromer (OST) Attacks

Our first example of access-driven cache attack was developed by Osvik, Shamir, and Tromer. They described and simulated several different methods on AES to perform local cache attacks. We cover only their most powerful attack in this chapter. The principle of their attack is very similar to the above basic model. They apply

the same idea on AES to determine which parts of AES tables are accessed during an AES operation.

In their attack, the adversary runs a spy process, which reads a local array to load it into the cache. After the spy process sets the state of the cache to a known state by forcing the CPU to replace the cache entries with the data of this array, the adversary triggers an AES encryption with a known plaintext. Immediately after the execution of AES, the spy process takes over the CPU and starts reading the same array again. However, this time it also measures the time it takes to read the blocks of this array. That way, as explained above, the adversary can determine which parts of the array got evicted from the cache. The parts of this local array that are not evicted by AES process directly give out which blocks of AES tables were not accessed during the encryption.

Let us consider the index $P_0 \oplus K_0$ as an example. This attack is a known plaintext attack and we know the values of P_0 but trying to find K_0. When we apply this attack, we determine the elements of AES tables that are not accessed during the encryption of this plaintext, i.e., the values that $P_0 \oplus K_0$ cannot be equal to. We can then eliminate the values of K_0 that would cause an access to these unaccessed elements for this particular value of P_0. If we can gather enough data from several encryptions, we end up with the correct value of $\langle K_0 \rangle$. Remember that we cannot reveal the least significant part of K_0 for the reason explained in the previous section. In general, if we consider the first round indices, we can reveal $\langle K_i \rangle$ for $0 \leq i \leq 15$. If we extend our focus on the second round indices, then we can recover the entire key.

Osvik et al. described several variations of access-driven cache attacks on AES. They also suggested relying on hardware-assisted SMT feature to detect the changes of the cache states on-the-fly during the encryption. However, their attack idea does not require SMT feature and can principally work on any multitasking environment.

Figure 18.5 shows real experimental results of an access-driven variant taken from [29]. In this experiment, Osvik et al. observed the cache activity of several encryptions with random but known plaintext values. They ran the spy process and collected "measurement scores" for each possible value of $\langle K_i \rangle$ as the following. They collected samples of the form (P, y, m) consisting of arbitrary table indices y, random plaintexts P, and measurement scores m. The measurement scores are the time delays when the spy reads the blocks of its local array that map to the

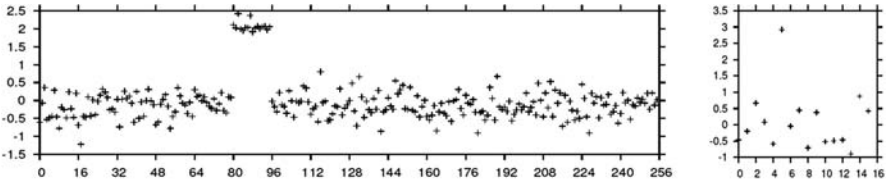

Fig. 18.5 Average measurement scores of first round OST attack for 30,000 (left) and 800 (right) triggered encryptions under the same key. The x-axis shows $P_5 \oplus y$ (left) and $\langle P_5 \oplus y \rangle$ and the y-axis shows the average measurement scores in units of clock cycles. The actual value of $\langle K_5 \rangle$ can easily be determined as 0×5.

same cache set as the block of the AES table that contains table element with the index y. If we consider a key byte $\langle K_i \rangle$, whenever $\langle y \rangle$ becomes equal to $\langle P_i \oplus K_i \rangle$ for a particular P, the measurement score m will be higher for this (P, y) pair. If we collect measurement scores (P, y, m) for a sample of known plaintext, split these scores into different subsets based on $\langle P_i \oplus y \rangle$ values and calculate the average m values in each of these subsets, then the subsets that correspond to the correct $\langle K_i \rangle$ value will have higher average measurement scores compared to the other subsets. This is exactly what Osvik et al. did to generate the results shown in Figure 18.5.

18.3.7 Percival's Hyper-Threading Attack on RSA

Percival developed a cache attack on RSA, which relies on hardware-assisted SMT capability [35]. Our attack model described in Section 18.3.6 can work on almost any system. But, such access-driven attacks become much more powerful on simultaneous multithreading environments, because the adversary can run the spy process simultaneously with the cipher. Running these processes simultaneously allows an attacker to obtain not only the set of data structures accessed by the cipher but also the approximate time that each access occurs.

In Percival's attack, the adversary again runs a spy process but this time it is run simultaneously with the server on an SMT processor. Spy process has a local array just like the previous attack and continuously reads each block of this local array in the same order. Note that each of these blocks correspond to a different cache line. The spy process reads the blocks that map to the same cache set together and measures the overall read time for each of these sets. In other words, the spy process observes each cache set (via reading the local array in a structured manner) in a certain order to determine whether the RSA process modifies this cache set. If reading a cache set takes longer, the attacker can conclude that this set was accessed during the time interval between the last read of the set by the spy process and the current read.

The experimental results of Percival's attack are given in Figure 18.6. The color of each little square in this figure indicates the time it takes the spy to read the corresponding cache set, denoted as cache congruency class. These colors can be considered as the measurement scores for each cache set. All of the squares in a particular column map to the same cache set. The vertical axis shows the time of the observation.

In general, such an attack can reveal the "footprints" of a victim process. Percival applied this idea on RSA and was able to identify the order of squaring and multiplication operations in OpenSSL's RSA implementation. Percival's attack on OpenSSL's sliding windows exponentiation (with a window size of 5 bits) could reveal an average 200 bits of information about each of the two 512-bit secret exponents.

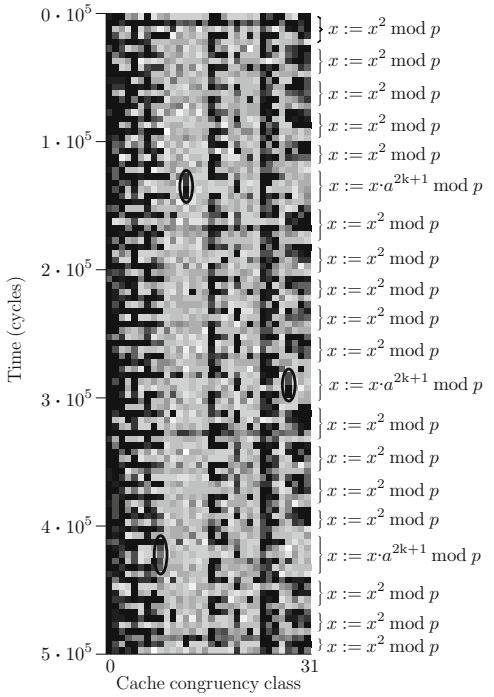

Fig. 18.6 Graphical representation of a small portion of the spy measurements in Percival's attack on RSA.

18.4 Branch Prediction Analysis

Branch prediction analysis (BPA) is the second microarchitectural analysis type we cover in this chapter. Several methods to exploit the branch prediction mechanism are developed [3, 4]; branch prediction mechanism is nowadays a part of all general purpose processors. We call all of these attacks branch prediction analysis. The most powerful variant of BPA is called simple branch prediction analysis (SBPA) and it is our subject in this section. Please refer to [4] for other variants of BPA.

18.4.1 The Concept of Branch Prediction

Deep microprocessor pipelines coupled with the ability to fetch and issue multiple instructions at every machine cycle led to the development of superscalar processors. Superscalar processors target a theoretical or best-case throughput of less than one machine cycle per completed instructions, cf. [39]. However, the actual throughputs of superscalar processors are limited by the available instruction-level parallelism (ILP) in the executed code. When branch instructions were recognized as one

of the most crucial performance killers of superscalar processors, microprocessor architects quickly invented the concept of branch predictors in order to circumvent those performance bottlenecks. There has been very significant amounts of research on more and more sophisticated branch prediction mechanisms, cf. [34, 38, 39]. However, it turns out that the branch mechanisms can be exploited to attack the integrity of the processes running on the processor.

Superscalar processors rely on branch prediction mechanisms to execute instructions speculatively to overcome control hazards, cf. [34, 39]. Therefore, the actual performance of microprocessors is greatly affected by the efficiency of speculation, which makes it one of the key issues in modern superscalar processor design.

A *branch instruction* is a point in the instruction stream of a program where the next instruction is not necessarily the next sequential one. There are two types of branch instructions: unconditional branches (e.g., jump instructions, goto statements) and conditional branches (e.g., if-then-else clauses, for and while loops). For conditional branches, the decision to take or not to take the branch depends on some condition that must be evaluated in order to determine the correct execution path. During this evaluation period, the processors speculatively execute instructions from one of the possible execution paths instead of stalling and awaiting for the decision to come through.

The key to achieve higher processor performance is the ability to predict the correct execution path as accurately as possible. The ultimate goal of branch prediction mechanisms is therefore to predict the most likely execution path in such a case. The accuracy of branch prediction mechanisms greatly affects the overall performance. Thus, it is very beneficial if the branch prediction algorithm tries to predict the most likely execution path in a branch. If the prediction is true, the execution continues without any delays. However, if the speculatively executed instruction flow turns out to be wrong, which is called a *misprediction*, the instructions on the pipeline that were speculatively issued have to be dumped and the execution has to start over from the mispredicted instruction path, thus suffers from a misprediction delay. Measurable timing differences between a correct and incorrect prediction are exactly what the BPA/SPBA attacks capitalize on.

A microprocessor needs the following information to speculatively execute the instructions after a conditional branch:

- *The outcome of the branch prediction.* It has to determine the outcome of a conditional branch, i.e., whether it needs to be taken or not taken, in order to execute the correct instruction sequence. This outcome depends on the evaluation of the condition and it is not immediately available when a conditional branch is issued. The processor must execute the branch and check the outcome of the condition, which is evaluated in later stages of the pipeline. Instead of waiting the *actual* outcome of the branch to come through, the processor starts executing a possible instruction sequence, which is predicted as the correct sequence by the branch prediction unit (BPU). This prediction is usually based on the history of the same branch as well as the history of other branches executed just before the current branch, cf. [39].

- *The target address of the branch.* If the BPU predicts a conditional branch to be taken, the instructions in the target address have to be fetched and issued. In this case, the processor needs the address of the predicted instruction sequence, i.e., the target address, in order to fetch and issue these instructions. Similar to the outcome of the branch, the target address may not be immediately available too. Therefore, the CPU tries to keep records of the target addresses of previously executed branches in a buffer, the so-called branch target buffer (BTB).

Overall common to all *branch prediction units (BPU)* is the following Figure 18.7. As shown, the BPU consists of mainly two "logical" parts, the BTB and the predictor. The predictor is that part of the BPU that makes the prediction on the outcome of the branch under question. The BTB is the buffer where the CPU stores the target addresses of the previously executed branches. Since this buffer is limited in size, the CPU can store only a number of such target addresses, and previously stored addresses may be evicted from the buffer if a new address needs to be stored instead. If the processor cannot find a target address in BTB (called a BTB miss), the execution suffers from a BTB miss delay similar to a branch misprediction. Further details of branch prediction can be found in [34, 38, 39].

18.4.2 Simple Branch Prediction Analysis

Several methods to exploit the branch prediction mechanisms are developed [4]. One of these methods presented relies on the fact that processors keep the target addresses of recently executed conditional branches in BTB. This attack, which was initially named as "trace-driven BTB attack", was significantly improved in [3] and renamed as simple branch prediction analysis (SBPA).

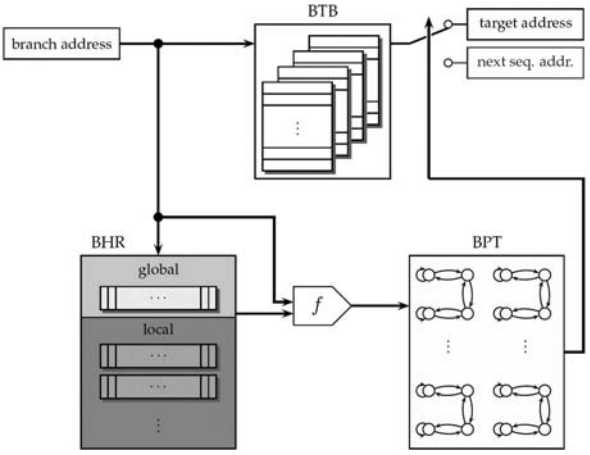

Fig. 18.7 Branch prediction unit architecture.

SBPA is also a spy-oriented attack similar to the cache eviction attacks presented above. However, there is a fundamental difference between SBPA attacks and pure data cache eviction attacks. Attacking the BTB is targeting the instruction flow, which is more complicated than the data flow within the memory hierarchy, i.e., between the L1 data cache and the main memory.

The SBPA attack is applied on RSA by running a spy process simultaneously with RSA on a multithreaded processor. It could reveal the execution flow of the RSA process by observing the BTB state transitions during a single RSA operation. Although the attack is carried out on an SMT system, it is argued that this attack can be used on almost any processor [7].

SBPA takes advantage of the fact that both processes share the BTB. The spy process continuously executes a certain fixed sequence of a sufficiently higher number of branches to guarantee the eviction of a target branch executed by the cryptographic process. In [3], the conditional branch of RSA, which determines the square and multiply sequence of the exponentiation, was targeted. When the next time the cryptographic process executes the target conditional branch, the address of this branch cannot be found in BTB. If the cipher takes this branch, the processor must rewrite the target address back to BTB, which causes the footprint of this branch to be left in BTB. Since the spy process continuously executes several branches, it will detect this change shortly after it happens. This way, the spy process can observe the traces of the target branch in terms of "taken" and "not taken".

We have to mention that the branches executed by the spy maps to the same BTB area with the target branch. In other words, the spy process intentionally creates conflicts (thus a race condition) between its branches and the target branch. Therefore, whenever the target branch turns out to be taken, the target address of this branch needs to be stored in BTB by evicting one of the spy branches from the BTB. When the spy process re-executes its branches, it will encounter a misprediction on the branch that has just been evicted. This misprediction will also trigger further mispredictions because the entry of the evicted spy branch needs to be re-stored and another not-yet-re-executed spy branch entry has to be evicted, which will also cause other mispredictions. Overall, the execution time of this spy step suffers from *many* misprediction delays resulting in a high timing gap between taken and not-taken situations of the target branch.

Reference [3] demonstrated the first SBPA attack on the S&M algorithm implemented in OpenSSL-0.9.7. They showed that measurements taken from a *single* run of the S&M exponentiation is sufficient to extract almost all of the RSA secret exponent bits. Figure 18.8 shows their experimental results. The y-axis shows the measurements of the spy in terms of the clock cycles and x-axis shows the order of these measurements, which also indicates the order of the RSA operations. As you can see, the spy measurements become clearly different when RSA executes a multiplication compared to the case of a square operation. Since the order of the spy measurements is also known, we can easily extract the operation sequence of RSA and construct the secret exponent from these measurement results.

Attacking the S&M algorithm was only a case study to show the potential of SBPA. The actual potential of SBPA is much broader as stated in [3, 7]. In general,

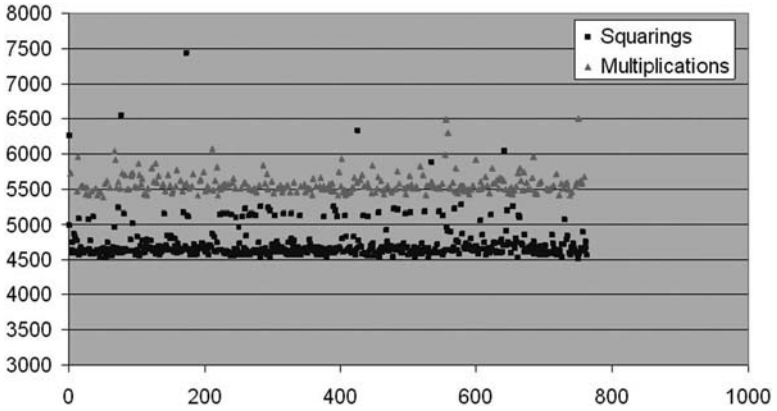

Fig. 18.8 Experimental results of SBPA on S&M exponentiation, yielding 508 out of 512 secret key bits.

SBPA can reveal the execution flow of a process and thus endangers any system if their execution flow depends on sensitive information. Several potential application areas of SBPA are given in [3]. Reference [7] identifies a novel side-channel attack on binary extended euclidean algorithm, which is enabled by the SBPA methodology.

18.5 I-cache Analysis

Instruction cache was identified as another microarchitectural analysis source [2]. This new technique, called instruction cache analysis or shortly I-cache analysis, is also spy oriented and tries to reveal the execution flow of a process just like branch prediction attacks.

Many processors use different caches for data and code segments of a process. Instruction cache (I-cache) stores recently executed instructions from the code segment and when a process starts executing a code block that is not currently stored in the cache, i.e., in case of a cache miss, the processor loads these instructions from main memory into the cache. Since a cache is limited in size, several different code blocks share the same cache sets/lines. In case of a cache conflict between different code blocks, they evict each other from the instruction cache when their executions are interleaved. As you can see, the functionality of instruction cache is very similar to data cache and also shared between different processes. Since we already explained above some details of cache architectures and the general functionality, which is very similar in both data and instruction cache, we do not cover these concepts again in this section.

In [2], the consequences of cache conflicts are exploited by creating intentional conflicts between the instructions of an RSA process and a spy code and forcing the

processor to evict the RSA instructions out of I-cache. The attack scenario given in [2] is the following.

A spy process runs simultaneously with the cipher process and tries to determine which instructions are executed by the cipher. It spies on the cipher execution by observing the I-cache state transitions. Assume that the spy process tries to understand when the cipher "touches" a certain I-cache set during the execution of a part of cipher code. The spy process continuously executes a set of "dummy" instructions that map to this particular cache set. These dummy instructions are not intended to perform any useful calculations other than filling some I-cache space, i.e., the "spied-on" cache set. These dummy instructions fill completely and precisely this I-cache set, no more, no less. Therefore, the processor has to store them into the spied-on cache set, which inevitably causes the eviction of the previous entries in that I-cache location and sets it to a known predetermined state. When the cipher executes some instructions that map to this particular I-cache location, the predetermined state set by the spy process must change. The spy process can determine this change via the timing difference when it re-executes its dummy instructions.

Reference [2] applied this basic principle to OpenSSL's RSA implementation. Due to some performance improvement reasons, OpenSSL first calls either multiplication or square functions from its multiprecision library during a Montgomery operation and then calls Montgomery reduction function to reduce the result to the modulus. This technique causes key-dependent sequence of multiplication and square function calls during sliding window exponentiation, which is the default exponentiation algorithm used in OpenSSL. In the experiments of [2], the spy continuously executed dummy instructions to evict the instructions of the multiplication function and measured the execution time of its own dummy instructions as described above. Note that, in this practical attack, the spy does not observe a single I-cache set but a number of sets that can hold the instructions of the multiplication function.

Figure 18.9 shows the results of this experiment. Again the y-axis shows the measurements of the spy in terms of the clock cycles and x-axis shows the order of

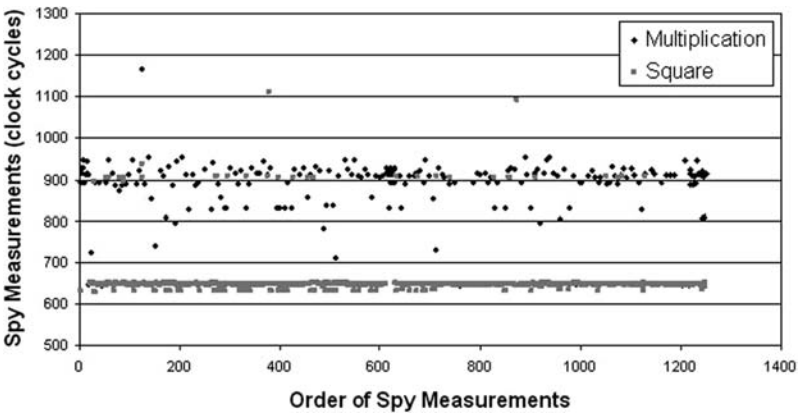

Fig. 18.9 Experimental results of I-cache analysis given in [2].

these measurements, which also indicates the order of the RSA operations. These timing measurements were taken during *a single RSA operation* under a random 1024-bit key. As you can see in this figure, the operation sequence of RSA could be successfully observed via I-cache analysis.

18.6 Exploiting Shared Functional Units

All of the microarchitectural analysis types we have covered above rely on the fact that there are some buffers (e.g., data cache, instruction cache, BTB) shared between different processes running on the same processor. The state of these buffers and the transitions between these states are affected by the execution of any of these processes and also affect the execution of other processes. These attacks can be applied on any platform with or without SMT capability as explained in [26]. However, [1] presents a novel attack which is very unique to certain SMT architectures and it seems that it cannot be carried out on CPU architectures without SMT hardware assistance. In this sense, this attack is unique because it does not rest upon a shared resource with the persistent state property between context/process switches. Instead, it is based upon the fact that some SMT technologies share complex functional units between the hardware-supported threads, i.e., between logical processors within a physical SMT processor, in order to keep the SMT area over head cheap.

Reference [1] presents an attack on OpenSSL which exploits the fact that the integer multiplier in Pentium 4 architecture is shared between the two threads executing on the same SMT-enabled processor. Since a multiplier does not preserve any persistent state, this attack methodology is quite different than other microarchitectural types. The principle idea of this attack is the following. A spy process continuously executes a number of dummy multiplication instructions and measures their execution time. Whenever the other process (RSA process in this case) performs some multiplications by executing its own multiplication instructions, the time measured by the spy will be longer. This is because the spy and the cipher instructions race to occupy the shared multiplier. When the multiplier executes a multiplication instruction from the cipher process, the spy multiplication instructions have to wait their turn until the multiplier finishes its current task, which eventually cause longer execution time for the spy instructions and can easily be detected by the spy.

Reference [1] also exploited the key-dependent RSA implementation of OpenSSL. As we already explained above, OpenSSL implementation has key-dependent sequence of multiplication and square function calls during sliding window exponentiation. We need to mention another important aspect of OpenSSL's implementation. RSA operations make use of multiprecision multiplication routines due to their long operand sizes. Usually the operands in RSA exponentiations are 512 bits or 1024 bits long, respectively, for 1024 and 2048 bit RSA keys. Note that RSA implementations usually benefits from Chinese remainder theorem and operates on half-sized operands compared to the size of entire public keys. Multiprecision libraries represent large integers as a sequence of machine-sized words.

OpenSSL implements two different multiplication algorithms: Karatsuba and "normal" multiplication. OpenSSL uses Karatsuba multiplication to multiply two numbers with an equal number of words (e.g., square operation). Karatsuba multiplication takes $O(n^{\log_2 3})$ time, where n is the number of words in the operands, cf. [25]. When multiplying two numbers with an unequal number of words of size n and m, OpenSSL executes normal multiplication, which runs in $O(nm)$ time. Therefore, a square operation takes less time than a multiplication in OpenSSL. This particular way of implementing RSA causes the leakage of operation sequence because the execution time variations depend on this sequence due to the difference between the implementations of multiplication and square operations.

Shared functional unit attack uses this timing difference, which can be observed by the spy process as described above, to distinguish between modular multiplications and squares. The operation sequence reveals the entire secret key in a binary square and multiply exponentiation. In case of OpenSSL's sliding window exponentiation, a large number of key bits can be derived.

18.7 Comparing Microarchitectural Analysis Types

Data cache attacks try to reveal the data access patterns of cryptosystems. On the other hand, branch prediction, I-cache, and shared function unit attacks expose the execution flow of the ciphers. Implementations with fixed instruction flow, which is usually the case for block ciphers, are intrinsically protected against these attacks. However, public key cryptosystem implementations, e.g., those of RSA and ECC, usually have key-dependent operation flow. It is possible to implement these systems without key-dependent execution flow, but it comes with some degree of performance loss. Due to such performance and optimization reasons, the developers usually choose to implement these systems in a way that cause execution flow variations, which make BPA, I-cache, and SFU attacks a real threat to actual security systems.

It is also possible to determine the execution flow of a cipher (e.g., RSA) by analyzing the data access patterns as done in [35]. However, implementations can avoid this threat by carefully handling the layout of data structures on the memory. For example, OpenSSL changed the way it handles the RSA structures to avoid data cache attacks. Even when the data structures are handled in a special way, BPA, I-cache, and SFU analysis can compromise the implementations if the execution flow remains key dependent. Similarly, data cache attacks can be applied on implementations with fixed execution flow if the data access patterns are key dependent. Therefore, both data and instruction cache analysis must be considered during the design and implementation of security critical systems.

The basic difference between I-cache and branch prediction analysis is the following. Branch prediction analysis presented in [3, 4] specifically targets conditional branches. A conditional branch controls the execution of different instruction paths. Thus, the outcome of a conditional branch, which can be observed via

BPA, leaks the instruction path to an adversary. However, using conditional branch is only one way to implement execution flow control. There are other techniques, which may be protected against BPA, to conditionally alter the execution flow without the use of conditional branches. In this sense, I-cache analysis is broader than BPA because it reveals the execution flow regardless of how execution flow control is implemented. For example, [8] proposes to use indirect jumps instead of conditional branches as a countermeasure to BPA, but this mitigation is still vulnerable to I-cache analysis.

The main difference between shared FU attack and the other MA types is the fact that it does not exploit the footprints of cryptosystems that are left on the persistent states of a buffer. Other MA types, data cache, I-cache, and BPA attacks rely on the persistent states of some buffers shared between different processes running on the same processor. These attacks can be applied on any platform with or without SMT capability. On the other hand, shared FU attack is based upon the fact that some SMT technologies share complex functional units between the hardware-supported threads and thus it seems that it cannot be carried out on CPU architectures without SMT hardware assistance.

18.8 Countermeasures for Microarchitectural Analysis

In this section we investigate possible countermeasures to prevent MA threats. We cover mainly software-based countermeasures and very briefly mention possible hardware-based countermeasures. The reason why we do not cover hardware-based approaches in greater length is because there had not been any real attempt, unfortunately, to employ such hardware changes in real systems. Therefore, we focus on more practical aspects of MA prevention, more specifically software mitigation methods, in this section.

Several countermeasures for AES were proposed in [13] against cache attacks. Particularly, [13] argues that permuting the AES lookup tables prevents access-driven attacks. They also propose to use smaller lookup tables, e.g., original AES S-box, during the first and last rounds of AES computations. Current cache attacks on AES exploit first, second, and the last round accesses. Using smaller tables during these rounds make it more difficult, i.e., require more samples, to apply these cache attacks. A formal study was presented in [41] to analyze the effects of table sizes, among other parameters, on the performance of time-driven attacks.

The only cache attack on RSA is from Percival [35]. He exploited the fact that OpenSSL implementation accesses different data structures during square and multiplication operations. In other words, Percival's attack can extract the operation sequence of RSA by tracing the cache activities, cf., Section 18.3.7. Moreover, it is also possible to determine which table entries are used in a multiplication step because table entries were stored in consecutive regions of memory and thus they map to different cache sets. Reference [14] proposes an implementation technique that does not have these weaknesses. To be more precise, [14] proposes to change

the memory layout of RSA exponentiation table and to interleave the table entries in memory instead of storing them in a consecutive manner. This way, each access to any table entry results in touching the same cache lines, which makes the accesses to different table entries indistinguishable. However, one still can observe the operation sequence of RSA due to the simple fact stated above. Therefore, [14] also suggests to employ fixed window exponentiation, which has a constant operation sequence. These countermeasures were implemented in OpenSSL as an optional protection mechanism, i.e., a user has the option to turn these mechanisms on.

Branch prediction and I-cache attacks exploit execution flows of cryptosystems. The best mitigation method for these attacks is to implement cryptosystems with fixed execution flows. Reference [7] analyzed the strength of OpenSSL's RSA implementation considering branch prediction analysis and detected several weak points that needed to be changed. Particularly, [7] suggested to remove several conditional branches that affect the strength of RSA implementation. OpenSSL team took these suggestions into consideration and modified the implementations. Reference [7] also proposes a new method to implement high-level execution flow of RSA without any variations. A similar proposal is also given in [21].

Another mitigation method for branch prediction vulnerabilities is given in [8]. They suggest to implement conditional branches via indirect branching. In other words, their method comprises storing the addresses of the branch legs in memory and loading the corresponding address into a register during runtime based on the evaluated condition of a conditional branch and using this register as the jump target. Since they do not consider to avoid execution flow variations, the proposed mitigation does not truly provide high security. Their protection can easily be overcome by I-cache attacks.

There are also several hardware countermeasures in the literature proposed to prevent microarchitectural attacks. We do not cover the details of hardware-based countermeasures in this book. The interested readers can refer to [10, 32, 33, 47].

18.9 Exercises

1. What is (are) the difference(s) between time-driven and access-driven cache attacks? Which one is more efficient and why?
2. What is (are) the fundamental difference(s) between data cache and instruction cache attacks?
3. Both I-cache analysis and simple branch prediction analysis (SBPA) can very effectively reveal the execution flow of a cryptosystem. For example, both attacks can extract the sequence of multiplication/square operations in an RSA exponentiation by observing only a single run of the cipher. However, I-cache analysis is more general than SBPA in the sense that I-cache analysis can compromise SBPA-resistant systems. What is the reason of this situation?
4. Which cryptosystems are susceptible to branch prediction analysis?

5. Why is it easier to carry out microarchitectural attacks on simultaneous multithreading processors?
6. Which microarchitectural attack type does seem to work only on simultaneous multithreading processors? Why?

18.10 Projects

1. There are some reference cache attack codes in [10] and [35]. Verify Bernstein's attack on AES and Percival's attacks on RSA using these reference codes.
   Bernstein's AES attack mimics a remote cache attack. In real remote attacks, the timing measurements must be obtained by a client and they also contain a lot of noise due to network delays, etc. However, in Bernstein's experiments, the encryption time is measured inside the crypto process, i.e., the server, instead of in the client. Modify Bernstein's reference code and measure the actual response times inside the client and verify that this attack becomes practically infeasible in a realistic remote attack.
   Percival's attack, the experiments need to be performed on a simultaneous multithreading processor. You can run these experiments on an HT-enabled Pentium 4. In order to synchronize spy and crypto processes, you can use `pthread` library.
2. There are several publications in the literature, cf. [36, 37, 46], that give the details of how one can extract the secret exponent in an RSA exponentiation by observing the occurrences of extra reduction steps in Montgomery multiplication during the exponentiation. In other words, if we can observe which multiplication/square operations perform an extra reduction step during an entire modular exponentiation with a secret exponent, we can extract the value of this secret exponent by using some statistical analysis techniques. These techniques can also tolerate some levels of errors in the observations. SBPA and I-cache analysis are useful tools to perform such observations to detect the occurrences of extra reduction steps. Study [36, 37, 46], find an RSA implementation that employs Montgomery multiplication with extra reduction step (OpenSSL v.0.9.8e would suffice), and try to apply SBPA and I-cache attacks on this implementation to detect the extra reduction steps and extract the secret exponent. Also analyze the error rates in the observations and their effect on the necessary sample size.
3. Branch prediction attacks have been demonstrated only on conditional branches so far. Avoiding conditional branches in cryptographic implementations are thought to prevent these attacks. However, there are still many open questions. For example, it may also be possible to compromise security systems by detecting when and which unconditional branches are executed during the course of a cryptographic operation. Such information can be useful if the implementation has data-dependent execution time and/or control flow variations. Another problem is due to error detection purposes. The software implementations, whether cryptographic operations or regular applications, have many conditional branches to detect run-time errors/anomalies. For example, a widely accepted convention

is to use conditional branches, e.g., if-then-else statements, to check the return values of functions/methods to detect run-time errors, unexpected results, etc. Although these conditional branches have a low probability to alter the execution flow, they can be exploited via branch prediction analysis. These aspects need further investigations.

References

1. O. Acıiçmez and J.-P. Seifert. Cheap hardware parallelism implies cheap security. *4th Workshop on Fault Diagnosis and Tolerance in Cryptography — FDTC 2007*, pp. 80–91, IEEE Computer Society, 2007.
2. O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pp. 11–18, ACM Press, 2007.
   Also available at: Cryptology ePrint Archive, Report 2007/164, May 2007.
3. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. On The power of simple branch prediction analysis. *2007 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'07)*, R. Deng and P. Samarati, editors, pp. 312–320, ACM Press, 2007.
   Also available at: Cryptology ePrint Archive, Report 2006/351, October 2006.
4. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pp. 225–242, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007, also available at: Cryptology ePrint Archive, Report 2006/288, August 2006.
5. O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on AES (Short Paper). 8^{th} *International Conference on Information and Communications Security — ICICS06*, P. Ning, S. Qing, and N. Li, editors, pp. 112–121, Springer-Verlag, Lecture Notes in Computer Science series 4307, 2006. Full version is available at: Cryptology ePrint Archive, Report 2006/138, April 2006.
6. O. Acıiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pp. 271–286, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.
7. O. Acıiçmez, S. Gueron, and J.-P. Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. 11^{th} *IMA International Conference on Cryptography and Coding*, S. D. Galbraith, editor, pp. 185–203, Springer-Verlag, LNCS 4887, 2007, also available at: Cryptology ePrint Archive, Report 2007/039, February 2007.
8. G. Agosta, L. Breveglieri, I. Koren, G. Pelosi, and M. Sykora. Countermeasures Against Branch Target Buffer Attacks. *4th Workshop on Fault Diagnosis and Tolerance in Cryptography — FDTC 2007*, pp. 75–79, IEEE Computer Society, 2007.

9. D. E. Bell and L. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corporation, 1973.
10. D. J. Bernstein. Cache-timing attacks on AES. Technical Report, 37 pages, April 2005. Available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
11. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES power attack based on induced cache miss and countermeasure. *International Symposium on Information Technology: Coding and Computing - ITCC 2005*, vol. 1, pp. 4–6, 2005.
12. J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. *Cryptographic Hardware and Embedded Systems — CHES 2006*, L. Goubin and M. Matsui, editors, pp. 201–215, Springer-Verlag, Lecture Notes in Computer Science series 4249, 2006.
13. E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, February 2006.
14. E. Brickell, G. Graunke, and J.-P. Seifert. Mitigating Software Side Channels in AES and RSA Software. Developers track RSA 2006, RSA conference San Jose, 2006.
15. Department of Defence. *Trusted Computing System Evaluation Criteria (Orange Book)*. DoD 5200.28-STD, 1985.
16. R. C. Detmer. *Introduction to 80X86 Assembly Language and Computer Architecture*. Jones & Bartlett Publishers, 2001.
17. P. Genua. A Cache Primer. Technical Report, Freescale Semiconductor Inc., 16 pages, 2004. Available at http://www.freescale.com/files/32bit/doc/app_note/AN2663.pdf.
18. J. Handy. *The Cache Memory Book*. 2nd edition, Morgan Kaufmann, 1998.
19. NIST. History of Computer Security Project: Early Papers. National Institute of Standards and Technology, Computer Security Division: Computer Security Resource Center, available at http://csrc.nist.gov/publications/history/index.html
20. W. M. Hu. Lattice scheduling and covert channels. *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 52–61, IEEE Computer Society, 1992.
21. M. Joye and M. Tunstall. Securing OpenSSL Against Microarchitectural Attacks. *International Conference on Security and Cryptography — SeCrypt'07*, J. Hernando, E. Fernndez-Medina, and M. Malek, editors, pp. 189–196, INSTICC Press, 2007.
22. J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, vol. 8, pp. 141–158, 2000.
23. P. C. Kocher. Timing Attacks on Implementations of Diffie–Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology – CRYPTO '96*, N. Koblitz, editor, pp. 104–113, Springer-Verlag, Lecture Notes in Computer Science series 1109, 1996.
24. C. Lauradoux. Collision attacks on processors with cache and countermeasures. *Western European Workshop on Research in Cryptology — WEWoRC 2005*, C. Wolf, S. Lucks, and P.-W. Yau, editors, pp. 76–85, 2005.

25. A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1997.
26. M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. *13th International Workshop on Selected Areas of Cryptography — SAC'06*, E. Biham and A. M. Youssef, editors, pp. 147–162, Springer, Lecture Notes in Computer Science series 4356, 2007.
27. M. Neve, J.-P. Seifert, and Z. Wang. A refined look at Bernstein's AES side-channel analysis. *Proceedings of ACM Symposium on Information, Computer and Communications Security — ASIACCS'06*, p. 369, ACM Press, 2006.
28. D. A. Osvik, A. Shamir, and E. Tromer. Other People's Cache: Hyper Attacks on HyperThreaded Processors. Presentation available at http://www.wisdom.weizmann.ac.il/∼tromer/.
29. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology — CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, D. Pointcheval, editor, pp. 1–20, Springer-Verlag, Lecture Notes in Computer Science series 3860, 2006
30. R. van der Pas. Memory Hierarchy in Cache-Based Systems. Technical Report, Sun Microsystems Inc., p. 28, 2002, available at http://www.sun.com/blueprints/1102/817-0742.pdf
31. D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.
32. D. Page. Defending Against Cache Based Side-Channel Attacks. Technical Report. Department of Computer Science, University of Bristol, 2003.
33. D. Page. Partitioned Cache Architecture as a Side Channel Defence Mechanism. Cryptography ePrint Archive, Report 2005/280, August 2005.
34. D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. 4th edition, Morgan Kaufmann, 2006.
35. C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005. Available at http://www.daemonology.net/hyperthreading-considered-harmful/
36. W. Schindler. A Combined Timing and Power Attack. *PKC 2002*, D. Naccache and P. Paillier, editors, LNCS 2274, pp. 263–279, 2002.
37. W. Schindler and C. D. Walter. More Detail for a Combined Timing and Power Attack against Implementations of RSA. *9th IMA International Conference on Cryptography and Coding*, K. G. Paterson, editor, pp. 245–263, Springer-Verlag, LNCS Nr. 2898, 2003.
38. T. Shanley. *The Unabridged Pentium 4 : IA32 Processor Genealogy*. Addison-Wesley Professional, 2004.
39. J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, 2005.
40. O. Sibert, P. A. Porras, and R. Lindell. The Intel 80× 86 Processor Architecture: Pitfalls for Secure Systems. *IEEE Symposium on Security and Privacy*, pp. 211–223, 1995.
41. K. Tiri, O. Acıiçmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. *14th International Workshop on Fast Software*

Encryption — FSE 2007, A. Biryukov, editor, pp. 399–413, Springer, Lecture Notes in Computer Science series 4593, 2007.
42. Trusted Computing Group. http://www.trustedcomputinggroup.org.
43. Y. Tsunoo, T.Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. *Cryptographic Hardware and Embedded Systems — CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, editors, pp. 62–76, Springer-Verlag, Lecture Notes in Computer Science series 2779, 2003.
44. Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. *ISITA 2002*, 2002.
45. Y. Tsunoo, E. Tsujihara, M. Shigeri, H. Kubo, and K. Minematsu. Improving cache attacks by considering cipher structure. *International Journal of Information Security*, vol. 5(3), pp. 166–176, Springer-Verlag, 2006.
46. C. D. Walter and S. Thompson. Distinguishing Exponent Digits by Observing Modular Subtractions. *Topics in Cryptology — CT-RSA 2001, The Cryptographers' Track at the RSA Conference 2001*, D. Naccache, editor, LNCS 2020, pp. 192–207, 2001.
47. Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. *34th International Symposium on Computer Architecture — ISCA'07*, pp. 494–505, ACM Press, 2007.
48. W. Ware. Security Controls for Computer Systems. Report of Defense Science Board Task Force on Computer Security; Rand Report R609-1, The RAND Corporation, 1970.
49. Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide.
50. A. Seznec. Branch Prediction under Scrutiny for Possible Security Flaw available online at
http://www.irisa.fr/activity/new/007/branchpredictionattack004.