Yet Another MicroArchitectural Attack: Exploiting I-cache

Onur Acıiçmez

Samsung Information Systems America
Samsung Electronics R&D Center, San Jose, CA 95134, USA
onur.aciicmez@gmail.com

Abstract. MicroArchitectural Attacks (MA), which can be considered as a special form of Side-Channel Analysis, exploit microarchitectural functionalities of processor implementations and can compromise the security of computational environments even in the presence of sophisticated protection mechanisms like virtualization and sandboxing. This newly evolving research area has attracted significant interest due to the broad application range and the potentials of these attacks. Cache Analysis and Branch Prediction Analysis are the only types of MA that are currently known publicly. In this paper, we announce Instruction Cache (I-Cache) as yet another source of MA and present our experimental results which clearly prove the practicality and danger of I-Cache Attacks.

Keywords: Instruction Cache, Modular Exponentiation, Montgomery Multiplication, RSA, Side Channel Analysis, MicroArchitectural Analysis.

1 Introduction

Side-channel cryptanalysis exploit the information leakage from execution time, power consumption or any such side-channels during the computation of cryptographic operations, c.f. [16, 17]. Cryptographic implementations leak sensitive information because of the physical properties and requirements of the cryptographic implementations and computational environments. Classical cryptography analyzes the cryptosystems as perfect mathematical models and ignores such physical requirements, thus fails to identify side-channel leakages. Therefore it is inevitable to utilize both classical cryptography and side-channel cryptanalysis in order to develop and implement secure systems.

The initial focus of side-channel research was on smart card security. We discuss the reasons of this situation in the next section. For now, we just want to mention that side-channel analysis of computer systems started to attract more attention after Brumley and Boneh demonstrated a successful and practical remote timing attack on real applications over a local network [12]. Since then, we have seen increased research efforts on the security analysis of the daily life PC platforms from side-channel point of view. The most important recent advance in the field is the realization of MicroArchitectural Analysis, i.e., the side-channel security vulnerabilities due to the functionality of common processor components such as data cache and branch prediction unit, c.f. [2, 4, 3, 6, 24, 9, 27]. These advances initiated a

new research vector to identify, analyze, and mitigate the security vulnerabilities that are caused by the design and implementation of processor components.

All of these cited pure software MicroArchitectural attacks, including the one presented in this paper, can compromise security systems despite of sophisticated partitioning methods such as memory protection, sandboxing or even virtualization. The reason for the failure of these trust mechanisms is because these new attacks "*simply exploit deeper processor ingredients below the trust architecture boundary*" as stated in [2, 4].

It is crucial to identify every possible MicroArchitectural vulnerability in order to understand the real potential of MicroArchitectural Analysis and to develop more secure systems by employing appropriate software countermeasures and making required hardware changes to future processors. In this paper, we identify a new MicroArchitectural attack source, in other words, another processor component that cause security vulnerabilities. We show that Instruction Cache, which is used to reduce the average time to read instruction codes of the applications from main memory, can be exploited to extract sensitive information regarding the execution of a cryptosystem. Our results clearly show the practicality and danger of I-cache attacks and put I-cache into the list of known MicroArchitectural attack sources, which currently contains only data cache and branch prediction unit since these two are the only two components that are "publicly" known to cause MicroArchitectural vulnerabilities.

In the next section, we give an overview of MicroArchitectural Analysis including a brief history. We explain what an instruction cache is and how it works in Section 3. The basics of RSA cryptosystem and the details of OpenSSL's RSA implementation are presented in Section 4. Sections 5 and 6 outline the underlying idea of instruction cache attacks and detail a sample attack on sliding window exponentiation of RSA implemented in OpenSSL. We also present our experimental results, which proves the practicality of instruction cache analysis concept, in Section 6. Then we point out a protected RSA implementation and conclude our paper in the last section.

2 Overview and Brief History of MicroArchitectural Analysis

The initial focus of side-channel research was on smart card security. Smart cards store secret values and they are designed to protect and process these secrets. Therefore, there is a serious financial gain involved in cracking smart cards, as well as, analyzing them and developing more secure smart card technologies, and this is one of the main reasons why smart cards had the initial focus of side-channel research. However, the recent advances and trends in microprocessor market, especially the development of microprocessor based security features (e.g. Intel's LT and VT Technologies, AMD Pacifica), and also the recent promises from the Trusted Computing community indicate the security assurance of storing and processing secret values, establishing virtually separate execution environments, etc. on computer platforms. As an eventual result, the side-channel analysis of computer platforms has become a necessity.

Another reason of the high attention to side-channel analysis of smart cards is due to the ease of applying such attacks on smart cards. The measurements of side-channel information

on smart cards are almost "noiseless", which makes such attacks very practical. On the other hand, there are so many factors that affect such measurements on real commodity computer systems. These factors make it much more difficult to perform successful side-channel attacks on "real" computers within our daily life. Thus, the side-channel vulnerability of computer systems was not seriously considered to be harmful until 2003. This was changed when Brumley and Boneh demonstrated a successful and practical remote timing attack on real applications over a local network [12]. They simply adapted the attack principle introduced by Schindler in [29] and applied it to a real web server to show that side-channel attacks are a real danger not only to smart cards but also to widely used computer systems. Their work was significantly improved by Acıiçmez et. al. in 2005 [8].

We have seen increased research efforts on the side-channel analysis of commodity PC platforms for the last few years. Soon, it was realized that the functionality of some micro-processor components cause serious side-channel leakage. These efforts led to the development of MicroArchitectural Analysis area.

MicroArchitectural Attacks exploit the microarchitectural components of a processor to reveal cryptographic keys. The functionality of some processor components generates data dependent variations in execution time and power consumption during the execution of cryptosystems. These variations either directly gives the key value out during a single cipher execution (c.f. [2]) or leaks information which can be gathered during many executions and analyzed to compromise the system (c.f. [24, 9, 21]).

There are currently two types of MA in the literature: Cache Analysis and Branch Prediction Analysis. A cache-based attack, abbreviated to "cache attack" 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. The cache vulnerability of computer systems has been known for a long time, c.f. [15, 16, 18], 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 [24, 21, 27]. Furthermore, some of the cache attacks can even be carried out remotely, i.e., over a local network [6].

The previous cache attacks are data-path attacks, i.e., exploit the data access patterns of a cipher. The memory accesses of software cryptosystems, especially S-box based ciphers like DES and AES, employ key-dependent table lookups, indices of which are simple functions of the key and the plaintext. Revealing these memory access patterns, i.e. lookup indices, via cache statistics and the knowledge of the processed message, e.g. in a known-text attack, make it relatively easy to break these ciphers.

A new group of MA attacks, called Branch Prediction Analysis (BPA), on the other hand, exploit instruction path of a cipher [2, 1, 4, 3]. In other words, an adversary can reveal the execution flow of a cipher using BPA, and if this execution flow is key dependent as in the case of RSA and ECC, then he can compromise the system. The most powerful BPA, which is called Simple Branch Prediction Analysis (SBPA), was shown to extract almost all of the RSA key bits during a single RSA operation. Immediately after it became public around the end of 2006, SBPA attracted very significant attention due to its implications. Acıiçmez et. al. briefly

outlined why SBPA endangers most of the current systems and detailed some techniques to show how SBPA can be used to break even "thought-to-be-side-channel-immune" systems in [2, 1, 5].

In this paper, we combine these two concepts: exploiting cache architecture and revealing instruction paths. We present several attacks that rely on instruction cache (I-cache) architecture of CPUs. All of the previous cache attacks exploit the side-channel leakage through data cache of a CPU. To the best of our knowledge, this is the first approach to exploit side-channel leakage due to I-cache architectures. Our experimental results indicate that I-cache analysis is as efficient as SBPA.

3 Instruction Cache

Our software I-cache timing attack exploit the functionality of I-cache implemented in microprocessors. A high-frequency processor needs to retrieve 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 memory and the processor speed has been continuously increasing for the last 3 decades as Moore's Law holds. Common to all processors, the attempt to overcome the drawbacks of 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 is already in the cache. If the data is already in the cache (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 is 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 is 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.

Before moving on to the next section, we want to mention two very important concepts that affect the functional behavior of a cache: the mapping strategy and the replacement policy. We will only give very brief information on these concepts in this paper.For further discussion on cache architectures and locality properties see [14, 28, 13].

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

associative mapping. In a direct mapped cache, a particular data block can only be stored in a single certain location in the cache. 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 blend 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 (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 is most unlikely to be used in the near future. There are several cache replacement policies proposed in the literature (c.f. [14, 28]). 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 is least recently used among all of the candidate data blocks that can be evicted from the cache.

Many processors employ different caches for data and code segments of a process. The instruction cache is responsible for storing recently used instructions from the code segment and quickly delivering them to the processor core when the accessed instructions are in the I-cache. When a process starts executing a code block that is not in the cache, i.e., in case of a cache miss, the processor loads these instructions from main memory into the cache. This situation happens either at the initial execution of a function (i.e., a cold miss), or after a cache conflict (i.e., conflict miss). Since a cache is limited in size, several different code blocks share the same cache sets/lines. A cache conflict or collision is the situation that occurs when an attempt is made to store two or more different data/code items at a cache location that can hold only one of them. In case of a cache conflict between different code blocks, they evict each other from the instruction cache when their executions are interleaved. In our I-cache attacks, we exploit this particular consequence of cache conflicts by creating intentional conflicts between the instructions of RSA cipher and a spy code and forcing the processor to evict the RSA instructions out of I-cache.

4 Overview of RSA and OpenSSL implementation

RSA is the most widely used public key cryptosystem, strength of which comes from the hardness of factorization problem, cf. [20]. The main computation in RSA decryption/signing is the modular exponentiation $P = M^d (\mathrm{mod} N)$, where M is the message or ciphertext, d is the private key that is secret, and N is the public modulus. N is a product of two large primes p and q. If an adversary obtains the secret value d, he can read all encrypted messages and impersonate the owner of the key. Therefore, the usual main purpose of using timing attacks is to reveal this secret value. If the attacker can factorize N, i.e., he can obtain either p or q, from which the value of d can be easily calculated. Hence, the attacker tries to find either p, q, or d.

Since the size of an RSA key is large, e.g., 1024 bits, the exponentiation is (even on today's very powerful PC platforms) still very expensive in terms of the execution time. Therefore, actual implementations of RSA usually employ very efficient and optimized algorithms to accelerate the result of this operation.

The RSA implementation of OpenSSL, which is the most widely used open source cryptographic library, employs two different exponentiation algorithms depending on the user choice: sliding window and fixed window exponentiation algorithms, c.f.[23]. The option that is a little slower but protected against cache attacks use fixed-window exponentiation. In this method, the n-bit exponent d is considered to be in radix-2^b form, i.e. $d = (d_{k-1}, d_{k-2}, ..., d_0)_{2^b}$, where $n = k * b$. It requires a preprocessing phase to compute multiples of M so that many multiplications can be combined during the exponentiation phase. The steps of this algorithm are shown in Figure 1.

$e_1 = M$
for i **from** 2 **to** $2^b - 1$
    $e_i = e_{i-1} * M \ (mod\ N)$
$S = e_{d_{k-1}}$
for i **from** $k - 2$ **to** 0 **do**
    $S = S^{2^b} \ (mod\ N)$
    if $d_i \neq 0$ **then**
        $S = S * e_{d_i} \ (mod\ N)$
return S

Fig. 1. Fixed Window Exponentiation Algorithm

Sliding window algorithm is very similar to fixed window method, except a slight modification. In fixed window method the exponent d is split into consecutive windows of b consecutive bits. The number of multiplications can be further decreased by splitting d into odd windows of at most b consecutive bits, where the windows are not necessarily consecutive and may be separated by zero bits (cf. Figure 2). Here, d is considered to be in radix-2 form, i.e. $d = (d_{n-1}, d_{n-2}, ..., d_0)_2$.

Other than these exponentiation algorithms, both of the options make use of Montgomery multiplication and Chinese Remainder Theorem to increase the performance of the implementation. Montgomery Multiplication (MM) is the most efficient algorithm to compute a modular multiplication. It uses additions and divisions by powers of 2, which can be accomplished by shifting the operand to the right, to calculate the result; therefore it is very suitable for hardware architectures. Since it eliminates time consuming integer division operations during the calculation of modular reductions, the efficiency of the algorithm is very high.

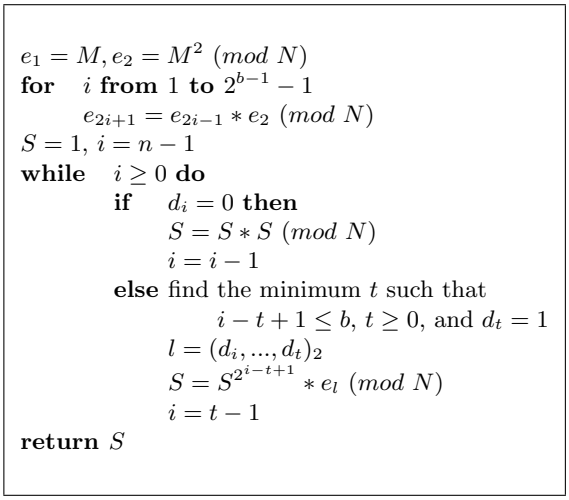

Fig. 2. Sliding Window Exponentiation Algorithm

Instead of computing $P = M^d \ (mod\ N)$ directly, OpenSSL use a more complex method to perform the same operation roughly 4 times faster. Modular exponentiation with full-length modulus N is replaced by two modular exponentiations with half-length modulus p and q and some additional operations to combine the results of these two independent exponentiations using Chinese Remainder Theorem.

OpenSSL also takes advantage of the difference between multi-precision multiplication and square operations to further enhance the performance. RSA data structures are much longer than regular machine word sizes and therefore span several machine words. Multi-precision libraries, i.e., the software libraries that handle long integer operations, represent large integers as a sequence of machine-sized (e.g. 32 or 64 bits) words. Intrinsically, the computation of a multi-precision square requires less number of word-wise multiplications compared to a multi-precision multiplication. OpenSSL has two different multiplication routines, Karatsuba and "normal" multiplication, and decides which one to use based on the operand sizes of the multi-precision operation. Karatsuba algorithm is used when the operands of the multiplication are composed of the same number of machine words. It has the time complexity $O(n^{1.58})$ where n is the number of words in the operand, cf. [20]. If the operand sizes are different, OpenSSL performs "normal" multiplication, which has the time complexity of $O(nm)$ where n and m are the numbers of words in the operands.

The implementation of OpenSSL utilizes these multi-precision multiplication algorithms for performance reasons. It takes advantage of the certainty of having two same size operands in square operations and handles the multiplication and square operations differently. During a montgomery operation, OpenSSL either calls multiplication or square functions from its multiprecision library, i.e., BIGNUM library, first and then performs montgomery reduction to reduce the result. In case of sliding window exponentiation, this technique cause key-

dependent sequence of multiplication and square function calls. We will explain in Section 6 how one can exploit this particular implementation detail of OpenSSL, i.e., key-dependent sequence of calls to multiprecision multiplication and square functions, to weaken the RSA system via I-cache attacks.

5 The concept of I-cache Attacks

Cryptosystems have data-dependent memory access patterns, which can be revealed by observing cache hit/miss statistics through side channels. Cache attacks rely on the cache hits and misses that occur during the encryption / decryption process of a cryptosystem. Even if the same instructions are executed for any particular (plaintext, cipherkey) pair, the cache behavior during the execution was shown to cause variations in the program execution time and power consumption. Cache attacks try to exploit such variations to narrow the exhaustive search space of cryptographic keys, c.f. [26, 32, 33, 6, 7, 24, 25, 9, 22, 19, 21, 27, 10, 11].

The previous studies in MicroArchitectural Analysis area, such as [24, 27, 21, 2, 4], initiated a new attack paradigm, which relies on simultaneous multi-threading / multi-tasking functionality of modern processors. Simultaneous multithreaded (SMT) processors have the capability of executing more than one execution thread simultaneously on the same physical processor. Multi-core processors can also execute more than one thread simultaneously. The main difference between multi-core and SMT is that expensive resources of a processor core (e.g., functional units, data and instruction cache, BTB) are shared between different threads in a SMT processor. The basic and simple resources are explicitly doubled to give the sense of two logical processor cores on a single physical processor core. Thus, this design technique enables the simultaneous execution of more than one process on the same physical processor core by taking advantage of thread-level parallelism, as if there were more than one processor [30, 31].

Single-threaded processor cores, on the other hand, executes only a single process/thread at any given time. However, the operating systems manage to distribute the processor time among all the active processes and give the users the feeling of a parallel, multi-threading execution. The OS basically decomposes the execution of each process into a series of short threads and schedules the execution of these threads with respect to each other.

Irrespectively of single-threaded or hardware-assisted multi-threaded, some processor resources are always shared among the active threads on the system, which enables one process to spy on another process, c.f. [24, 27, 21, 2, 4]. Although the memory protection mechanisms prevents a process to directly read other processes' data, the functionality of shared resources leak the so-called metadata, c.f. [24], and (e.g.) causes disclosure of the secret / private keys used in security systems.

In our I-cache attacks, we rely on the concept of executing a spy code, which keeps track of the changes in the state of I-cache, i.e., metadata, during the execution of a cipher process. A spy code / process can run simultaneously or quasi-parallel with the cipher process and determine which instructions are executed by the cipher. It achieves this goal by spying on the cipher execution via observing the I-cache state transitions.

Assume that an adversary tries to understand whether a certain I-cache set is "touched" by the cipher, i.e., modified, during the execution of a part of cipher code. The spy allows the cipher to run and takes over the processor shortly before the execution of the "spied-on" part of cipher code. This task of pausing the cipher execution at a determined point, even though it sounds nice, is very tricky and requires very fine-crafted spy code. However, it is feasible and was successfully used in earlier studies on cache attacks by Neve et. al. in [21] to devise an attack on the last AES round, which is composed of a small number of instructions.

After the spy takes over, it ensures that this particular I-cache set does not contain any instructions from the cipher by executing a set of "dummy" instructions. These dummy instructions are not intented to perform any calculations or tasks other than filling some I-cache space. These dummy instructions shall fill completely and precisely this I-cache set, no more no less. During the execution of dummy instructions, the processor has to store them into the cache, which inevitably causes the eviction of the previous entries in that I-cache location. That way, the spy sets the state of this particular I-cache set to a known predetermined state and then it lets the cipher run the "spied-on" part of code.

The spy takes control of the processor after the execution of a relatively small number of cipher instructions. If the executed cipher instructions touch the I-cache set under observation, this will cause the eviction of some of these spy-owned dummy instructions from the I-cache. When the spy takes control of the processor, it re-executes the same dummy instructions but this time also measures their total execution time. If some of these dummy instructions are not in I-cache, which indicates the modification of this I-cache set due to cipher execution, then the measured execution time will take longer simply because the evicted instructions must be retrieved from the memory which has a significantly larger latency compared to the cache.

During the quasi-parallel execution of spy and cipher processes, a malicious spy routine can continuously interrupt the cipher execution with short intervals and apply the above basic technique to every single I-cache set each time it takes the control. On simultaneous multi-threading systems, the spy routine does not even need to interrupt the cipher execution and can observe it "on-the-fly" as done in [27, 1]. If the adversary can get measurements with high enough resolution , i.e., if he can estimate which cache sets are modified during the execution of which part of the cipher code, this will reveal the execution flow of the cipher. Therefore, such a spy routine has the potential of revealing the entire execution flow of the cipher on almost any processor architecture as long as there is an I-cache and its metadata is preserved during the transfer of processor time between different processes.

In the next section, we will outline a sample cache attack on OpenSSL's sliding window exponentiation. We want to mention that we use OpenSSL's SWE just as a case study to prove the concept of I-cache analysis. The actual application range of I-cache attacks is much more broader than this simple case study. Our results from this case study show that an adversary can easily get a measurement resolution high enough to compromise very critical security systems.

6 A Case Study: Attack on OpenSSL's Sliding Window

As we mentioned above, OpenSSL takes advantage of the difference between multiprecision multiplication and square operations to improve the performance of its RSA implementation. During a montgomery operation, OpenSSL first calls either multiplication or square functions from BIGNUM library and then reduces the result to the modulus via montgomery reduction function. In case of sliding window exponentiation, this technique causes key-dependent sequence of multiplication and square function calls. The current version of OpenSSL employs either sliding window or fixed window exponentiation depending on the user's choice.

A practical method to reveal the multiplication/square operation sequence is the following. The spy function can evict the instructions of multiplication function (or square function, resp.) and measure the execution time of its own dummy instructions as described in the previous section. The higher execution time in spy measurements indicate the execution of this multiplication (square, resp.) function. Therefore, the spy can determine when the cipher calls this particular function, which also directly reveals the multiplication/square operation sequence. The spy function can perform the attack either "on-the-fly" on simultaneous multithreading systems (c.f. [27, 1])or via exploiting OS-scheduling (c.f. [21]).

Our attack scenario is the following. A "protected" crypto process executes the RSA signing/decryption process and also a spy process is executed simultaneously or quasi-parallel with the cipher and it continuously does the following:

1. continuously executes a number of dummy instructions, and
2. measures the overall execution time of all of these instructions

in such a way that these dummy instructions precisely evicts the instructions of BIGNUM multiprecison multiplication function from I-cache.

Assume that the multiplication function instructions span from logical address A to B. Due to the properties of cache architectures, an instruction block, i.e., continuous consecutive instructions, must span from logical address A_1 to B_1 to map to the same I-cache sets with the multiplication function, where the least significant parts of A and A_1 and also B and B_1 must be equal. To completely evict the multiplication function from I-cache, the spy routine has to execute a number of different such instruction blocks and this number needs to be equal or greater than the number of associativity of the I-cache. Clearly, implementing this spy routine requires the knowledge of logical address space of the multiplication function and the details of the I-cache. We want to mention that it is easy to learn the properties of an I-cache either from the manufacturer specs or by using simple benchmarks as explained in [34].

6.1 Experimental Details and Results

To validate our aforementioned I-cache analysis strategy, we performed some practical experiments. We compiled the RSA decryption function of OpenSSL (version 0.9.8d) with the choice of SWE Exponentiation. We disassembled the executable file to see the logical addresses of

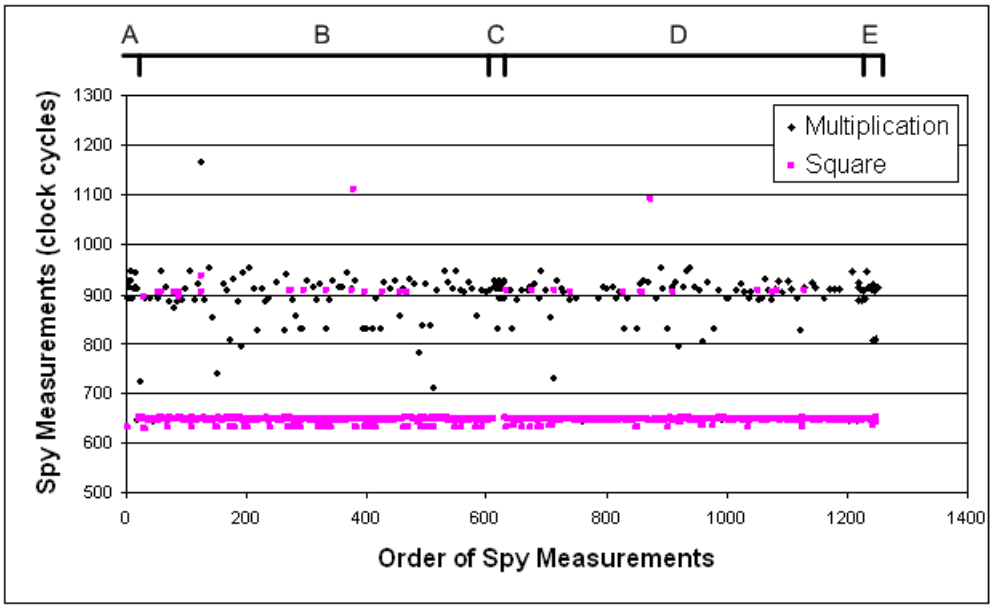

Fig. 3. Experimental Results

BIGNUM multiprecision multiplication function instructions. GNU Project debugger (i.e. gdb) has two functions, "info line" and "disas", that we used for this task. Then we implemented our spy routine based on these logical addresses and the parameters of the I-cache architecture on our platform. Then we carried out our attack by letting the spy routine run and make measurements with relatively short intervals during the execution of RSA signing operation.

After analyzing the spy measurements, we ended up with the results shown in Figure 3. These timing measurements are taken during *a single RSA operation* under a random 1024-bit key. We could reveal the operation sequence of RSA almost completely during a single RSA operation via I-cache analysis, as also achieved by Acıiçmez et. al. using SBPA. Revealing the multiplication / square sequence of an RSA with binary exponentiation algorithm would directly give the value of d out. In case of sliding window exponentiation, Percival stated that one can obtain around 200 bits of each 512-bit exponents when the operation sequence is known [27].

Looking closely to this graph, one can also distinguish different phases of the RSA execution:

A) table initialization phase of the first exponentiation
B) the first exponentiation phase
C) table initialization phase of the second exponentiation
D) the second exponentiation phase

E) the remaining RSA operations including CRT combination

If an adversary runs a stand-alone spy process on a machine, he also needs to ensure that the measurement results from the spy indeed correspond to the cipher process. Since there are possibly many other processes running on the same machine, he needs to distinguish when the spy process is measuring the cipher execution but not another process. In an experimental setup, we can set the system to ensure this synchronization by (e.g.) assigning high priorities to spy and cipher processes. However, in a real attack, where the adversary does not have privileges to configure such parameters, this synchronization issue becomes problematic. As seen in our results, different RSA phases are distinguishable, which may enable an adversary to understand whether the spy process is spying on a cipher process or another process and thus overcome the synchronization problem.

Our results clearly indicate that such I-cache attacks are as dangerous and serious as Simple Branch Prediction Analysis, which attracted very significant attention immediately after its publication because of the several very serious security vulnerabilities it implied. The same vulnerabilities sketched out in [2, 1] can also be exploited via I-cache analysis.

7 Conclusions

We have showed that a major processor component, Instruction Cache (I-cache), cause serious security vulnerabilities and can be used in a side-channel attack as a source of information leakage. The special side-channel area that exploits processor components is called MicroArchitectural Analysis (MA) and there are currently two types of MA in the literature: Cache Analysis and Branch Prediction Analysis. Our contribution in this paper is to introduce I-cache Analysis as yet another MA type.

We have presented a sample pure software-based I-cache attack on OpenSSL's RSA implementation and proved that I-cache attacks have the potential to reveal the execution flow of cryptosystems like RSA, which can lead to a complete break if the cryptosystem is implemented in a way to have key-dependent execution flow. Therefore, it is desirable to implement cryptographic software applications without any key-dependent instruction paths. A method was already outlined in [5] to protect RSA against instruction path attacks like Branch Prediction and I-cache Analysis. Such techniques and countermeasures must be developed for other cryptosystems and employed in cryptographic applications / libraries.

I-cache attacks are instruction-path attacks unlike the previous cache attacks, which exploit the data-path of a cipher execution. Branch Prediction Analysis is the first instance of MA that reveals the instruction paths of the cryptosystems. Similar to Simple Branch Prediction Analysis, the most powerful Branch Prediction Analysis method, I-cache analysis has the potential to reveal the complete operation sequence of a cryptosystem during a single execution. Simple Branch Prediction Analysis attracted very significant attention immediately after its publication because of the several very serious security vulnerabilities it implied. The same vulnerabilities pointed out in [2, 1] can also be exploited via I-cache analysis.

I-cache analysis, just like cache and branch prediction analysis, can compromise security systems even in the presence of security mechanisms like sandboxing and virtualization because all of these attacks exploit deep processor functionalities which are below the trust architecture boundary of these security mechanisms.

It is extremely important and urgent to identify every possible MicroArchitectural vulnerability in order to understand the real potential of MicroArchitectural Analysis and to develop more secure systems by developing and employing appropriate software countermeasures and possibly making required hardware changes to future processors.

References

1. 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), Singapore, March 20-22, 2007, to appear.
2. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. On The Power of Simple Branch Prediction Analysis. Cryptology ePrint Archive, Report 2006/351, October 2006.
3. 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, pages 225-242, Springer-Verlag, Lecture Notes in Computer Science **series 4377**, 2007.
4. O. Acıiçmez, J.-P. Seifert, and Ç. K. Koç. Predicting Secret Keys via Branch Prediction. Cryptology ePrint Archive, Report 2006/288, August 2006.
5. O. Acıiçmez, S. Gueron, and J.-P. Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. Cryptology ePrint Archive, Report 2007/039, February 2007.
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, pages 271-286, Springer-Verlag, Lecture Notes in Computer Science **series 4377**, 2007.
7. O. Acıiçmez and Ç. K. Koç. Trace-Driven Cache Attacks on AES. Cryptology ePrint Archive, Report 2006/138, April 2006.
8. O. Acıiçmez, W. Schindler, Ç. K. Koç. Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations. *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security*, C. Meadows and P. Syverson, editors, pages 139-146, ACM Press, 2005.
9. D. J. Bernstein. Cache-timing attacks on AES. Technical Report, 37 pages, April 2005. Available at: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
10. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. *International Symposium on Information Technology: Coding and Computing - ITCC 2005*, volume 1, pages 4-6, 2005.
11. J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. *Cryptographic Hardware and Embedded Systems — CHES 2006*, L. Goubin and M. Matsui, editors, pages 201-215, Springer-Verlag, Lecture Notes in Computer Science **series 4249**, 2006.
12. D. Brumley and D. Boneh. Remote Timing Attacks are Practical. *Proceedings of the* 12^{th} *Usenix Security Symposium*, pages 1-14, 2003.
13. 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
14. J. Handy. *The Cache Memory Book*. 2^{nd} edition, Morgan Kaufmann, 1998.
15. W. M. Hu. Lattice scheduling and covert channels. *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52-61, IEEE Computer Society, 1992.
16. P. C. Kocher. Timing Attacks on Implementations of Diffie–Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology - CRYPTO '96*, N. Koblitz, editor, pages 104-113, Springer-Verlag, Lecture Notes in Computer Science **series 1109**, 1996.

17. P. C. Kocher, J. Jaffe, B. Jun. Differential Power Analysis. *Advances in Cryptology - CRYPTO '99*, M. Wiener, editor, pages 388-397, Springer-Verlag, Lecture Notes in Computer Science **series 1666**, 1999.
18. J. Kelsey, B. Schneier, D. Wagner, C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, volume 8, pages 141-158, 2000.
19. 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, pages 76-85, 2005.
20. A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press, New York, 1997.
21. M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. *Selected Areas of Cryptography — SAC'06*, to appear.
22. 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*, to appear, Taipei, Taiwan, March 21-24, 2006.
23. Openssl: the open-source toolkit for ssl/tls. Available online at: `http://www.openssl.org/`.
24. 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, pages 1-20, Springer-Verlag, Lecture Notes in Computer Science **series 3860**, 2006
25. 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/`.
26. 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.
27. C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005. Available at: http://www.daemonology.net/hyperthreading-considered-harmful/
28. D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. 4^{th} edition, Morgan Kaufmann, 2006.
29. W. Schindler. A Timing Attack against RSA with the Chinese Remainder Theorem. *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç.K. Koç and C. Paar, editors, pages 110–125, Springer-Verlag, Lecture Notes in Computer Science **series 1965**, 2000.
30. J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.
31. A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating system concepts*. 7^{th} edition, John Wiley and Sons, 2005.
32. Y. Tsunoo, T.Saito, T. Suzaki, M. Shigeri, 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, pages 62-76, Springer-Verlag, Lecture Notes in Computer Science **series 2779**, 2003.
33. Y. Tsunoo, E. Tsujihara, K. Minematsu, H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. *ISITA 2002*, 2002.
34. K. Yotov, S. Jackson, T. Steele, K. Pingali and P. Stodghill. Automatic Measurement of Instruction Cache Capacity. 18^{th} International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2005.