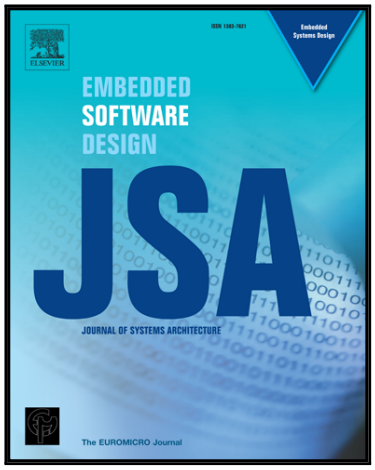
FLUSH + PREFETCH: A Countermeasure Against Access-driven Cache-based Side-Channel Attacks

M Asim Mukhtar, Maria Mushtaq, M Khurram Bhatti, Vianney Lapotre, Guy Gogniat

Journal Pre-proof



FLUSH + PREFETCH: A Countermeasure Against Access-driven Cache-based Side-Channel Attacks

M. Asim Mukhtar, Maria Mushtaq, M. Khurram Bhatti,
Vianney Lapotre, Guy Gogniat

PII: S1383-7621(19)30505-3
DOI: https://doi.org/10.1016/j.sysarc.2019.101698
Reference: SYSARC 101698

To appear in: *Journal of Systems Architecture*

Received date: 18 March 2019
Revised date: 23 October 2019
Accepted date: 30 November 2019

Please cite this article as: M. Asim Mukhtar, Maria Mushtaq, M. Khurram Bhatti, Vianney Lapotre, Guy Gogniat, FLUSH + PREFETCH: A Countermeasure Against Access-driven Cache-based Side-Channel Attacks, *Journal of Systems Architecture* (2019), doi: https://doi.org/10.1016/j.sysarc.2019.101698

FLUSH+PREFETCH: A Countermeasure Against Access-driven Cache-based Side-Channel Attacks

M. Asim Mukhtar[a,*], Maria Mushtaq[b], M. Khurram Bhatti[c], Vianney Lapotre[d], Guy Gogniat[e]

[a] *Information Technology University (ITU), 4th Floor, ASTP, 346-B, Ferozepur Road, Lahore, Pakistan, E-mail: mh.asim.mukhtar@gmail.com*
[b] *University of South Brittany, Rue Saint-Maude, Lorient, France, E-mail: maria.mushtaq@univ-ubs.fr*
[c] *Information Technology University (ITU), 4th Floor, ASTP, 346-B, Ferozepur Road, Lahore,Pakistan E-mail: khurram.bhatti@itu.edu.pk*
[d] *University of South Brittany, Rue Saint-Maude, Lorient, France, E-mail: vianney.lapotre@univ-ubs.fr*
[e] *University of South Brittany, Rue Saint-Maude, Lorient, France, E-mail: guy.gogniat@univ-ubs.fr*

Abstract

Cache-based side-channel attacks (SCAs) are becoming a security threat to the emerging computing platforms. To mitigate these attacks, numerous countermeasures have been proposed. However, these countermeasures require either radical hardware modification or they are incompatible with the performance features like super-page and data de-duplication. This paper presents a countermeasure, called Flush+Prefetch, which obfuscates the memory access behavior of a secure application using independent threads that randomly access the memory belonging to secure application. Unlike existing state-of-the-art countermeasures, Flush+Prefetch works with commodity hardware and it is compatible with existing performance features. As a proof-of-concept, we have studied the effectiveness of Flush+Prefetch by defending the secret key of RSA cryptosystem against a high-resolution cache side-channel attack called Flush+Reload. We have evaluated the confidentiality of RSA decryption process on an Intel Xeon E5-2643 processor by generating $100,000$ requests to a web-server sequentially while considering the effect on performance as well. Our experimental results show that the confidentiality of memory accesses by RSA is preserved under Flush+Prefetch countermeasure. Our results show that the performance, in terms of average execution time, is improved by 10.2% for best design case as compared to the system under attack.

Keywords:
Side-Channel Attacks (SCAs); Access Driven Cache-based SCAs; Obfuscation based Countermeasure; Noise; Flush+Reload; RSA.

1. Introduction

Cloud computing is an emerging computing paradigm in which multiple virtual machines (VMs) are developed on a same physical machine [1, 2]. Each virtual machine gets an abstraction of the shared physical machine provided by underlying layer known as virtual machine monitor (VMM). VMM takes care of isolation and exclusive resources between the virtual machines. The magnitude of sharing of physical hardware between virtual machines is fundamental to cloud economy but it raises security issues [3, 4, 5, 6, 7]. The most recent attacks known as *Side-Channel Attacks* (SCAs) exploit the shared physical resources even in existence of strict VMM isolation or theoretically secure cryptographic ciphers.

Access-driven cache-based side-channel attacks are an important class of cache based side channel attacks. These attacks have gained interest in research community because these can break popular cryptosystems with high accuracy. For example, Yarom et al. [8] have presented the Flush+Reload attack that retrieves up to 96.7% of secret key by observing the sequence of memory accesses made by RSA. RSA is considered to be theoretically secure cipher and its strength depends on the factorization of two large prime numbers [9]. Cache-based SCAs are of particular concern for modern security efforts in classical computer architectures that run cryptosystems as they are capable of successfully retrieving the secret cryptographic information *without* the knowledge of factorization of prime number.

Recently, various software and hardware based countermeasures have been proposed against cache-based SCAs [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. However, these countermeasures require radical hardware modifications or these are incompatible with the performance features like super-pages, data de-duplication and simultaneous-multithreading. For example, Shi et al.[13], Kim et al. [14] and Ji et al. [21] have proposed partitioning of cache using software-based techniques in order to mitigate cache-based SCAs, but these are incompatible with the super-pages [22], which yields slowing down of application by up to 10% to 33% [23]. Also, Liu et al. [18] have proposed randomization of memory-to-cache mappings in order to mitigate cache-based SCAs, but this countermeasure requires radical modifications in hardware, which unfortunately have not being adopted in mainstream processors. Thus, it is imperative to design a countermeasure against cache-based SCAs that works on commodity hardware and does not disable the widely used

*Corresponding author.
[1] Declarations of interest: none

performance features such as super-pages, data de-duplication, and simultaneous-multithreading, along with the strong security guarantees.

Our proposed countermeasure that we call Flush+Prefetch works on commodity hardware without disabling performance features. The proposed countermeasure requires minor modification at application level and works in multi-threading environment. This countermeasure takes benefit from two limitations of software-based cache attacks: 1) the attacks cannot identify the source that has generated a particular cache access and 2) These cannot detect multiple operations on a particular cache line. These limitations are exploited by injecting *noise* in cache access pattern through the use of concurrent threads that contain prefetch or clflush instructions. Doing so randomly encodes cache access pattern such that the attacker cannot extract the encryption/decryption key from cache access information. As a proof-of-concept, we have analyzed the security of RSA against Flush+Reload attack.

This paper makes the following contributions:

- We have designed and implemented two obfuscation mechanisms called as Flush+Prefetch, integrated with application that are independent to applications execution path in order to mitigate access-driven cache side-channels attacks. Flush+Prefetch requires fewer software modification and can execute on commodity hardware without disabling hardware performance features like super-pages, data de-duplication and simultanueous-multithreading.
- We have evaluated the security of these mechanisms by defending the secret key of RSA cryptosystem against a high-resolution cache side-channel attack called Flush+Reload attack. We have analyzed 100, 000 memory access traces of RSA in presence of Flush+Prefetch countermeasure to show the confidentiality of secret key.
- We have evaluated the performance overhead of Flush+Prefetch countermeasure for both mechanisms and find that performance overhead is smaller than previous state-of-the-art single path programming based countermeasure [24].

The remainder of this paper is organized as follows. Section 2 provides related work and background on cache-based side channel attacks, countermeasure techniques and Flush+Reload attack. Section 3 presents our proposed Flush+Prefetch countermeasure technique for SCAs. Section 4 presents the results of our experimental evaluation in detail. Section 5 compares our countermeasure with other application level countermeasures. Section 6 discuses the synchronization and generalization aspect of our countermeasure. Finally, Section 7 concludes this paper.

2. Related Work & Background

In this section, we provide overview of cache based SCAs and related countermeasures, and background on Flush+Reload attack and RSA cryptographic cipher.

2.1. Cache-Based SCAs and Countermeasures

As caches are shared among processes, execution time of co-running processes varies based on contention among cache lines. By carefully orchestrating the contention, memory access trace of any co-running processes can be extracted, such attacks are known as access-driven cache based-side channel attacks. These attacks have been extensively studied in past few decades [25, 26, 8, 3, 27, 28, 29, 30, 31, 32]. Initially, the attacks were presented on single core architecture, which required to compromise OS scheduler for successful attack [25, 28, 29, 30]. Recently, these attacks have been presented on multi-core architecture without the assumption of compromising OS/VMM, which is a more realistic assumption [26, 8]. Among these, Flush+Reload attack is the fastest and accurate attack and have been raising high-security threat when it is combined with other microarchitectural vulnerabilities such as demonstrated in Meltdown [33] and Spectre [34] attacks.

To mitigate these attacks, numerous countermeasures have been proposed by modifying the components of system software, hardware or applications [22, 13, 14, 15, 21, 10, 35, 24]. At system and hardware -levels, countermeasures mainly modify the page allocation algorithm or cache controllers to eliminate sharing among processes. These countermeasures require radical modifications and do not guarantee compatibility with features such as data-deduplicaiton, superpages and simultaneous multithreading (SMT) [22]. Therefore, these countermeasures have not being adopted in mainstream processors. As compared to these, modifications at application-level are relatively easy and flexible. Brickell et al. [10] have modified the AES implementation to introduce compaction, randomisation and preloading of lookup table for hiding the real memory access trace in noise. However, this implementation incurs high performance overhead of about 100% to 120% and it is specific to AES implementation. Crane et al. [35] used dynamic software diversity to obfuscate the cache access traces by randomly executing the multiple different copies of the security-critical part of the application. However, this countermeasure also incurs high performance overhead of about 76%. Rane et al. [24] have presented a compiler-based solution named as Raccoon that made memory access trace constant by modifying the execution behavior of secret dependent branches in an application. The performance overhead of Raccoon becomes significantly larger when number of branches increases. Most of the application-level countermeasures have reported high overhead because of extraneous execution path comes in the critical path of application. Unlike previous application-level countermeasure, Flush+Prefetch decouples the extraneous execution path from the critical execution path of application.

2.2. Flush+Reload Attack

In this article, we take Flush+Reload as a proof-of-concept to show the effectiveness of our countermeasure. Therefore, we discuss this attack in detail for an understanding of the following sections. The Flush+Reload attack fundamentally exploits the timing difference of cache *hit* as compared to cache *miss*. Figure 1 shows how the attack works. One round of

Flush+Reload attack technique consists of three distinct phases: Flush, Wait, and Reload as shown in Figure 1. In the first phase, attacker, while targeting shared Last-Level Cache (LLC), first evicts a shared cache line of interest using *clflush* instruction. [8]. In the second phase, attacker waits for a prefixed duration and allows victim to execute. In the last phase, attacker using *mov* instruction reloads the same cache line it had evicted earlier and measures its loading time using *rdtsc* instruction. A slow reload would indicate that attacker is the only process touching that particular cache line, thus every reload will cause a cache miss and will take more time to fetch data from main memory. A faster reload, however, indicates that the victim process also touched the cache line of interest while attacker was in his wait phase, thus a cache hit for attacker and it takes less time to fetch data from cache. Figure 1 shows various possibilities during an attack round. Case-A shows that the victim process does not access cache line at all, while Case-B shows the victim accessed once. Significant difference in the reload time is measured. Case-C shows the victim's access can overlap the reload phase of attacker, which allows victim to benefit from attacker's reload for execution. The attacker, in this case, reports a cache miss for himself. Case-D shows an opposite scenario when attacker benefits from victim's access. Case-E shows the likelihood of not missing on any access by the victim if attacker targets a cache line of interest belonging to loop body. Flush+Reload has high resolution as it targets specific cache lines that belong to operations of cryptosystem.

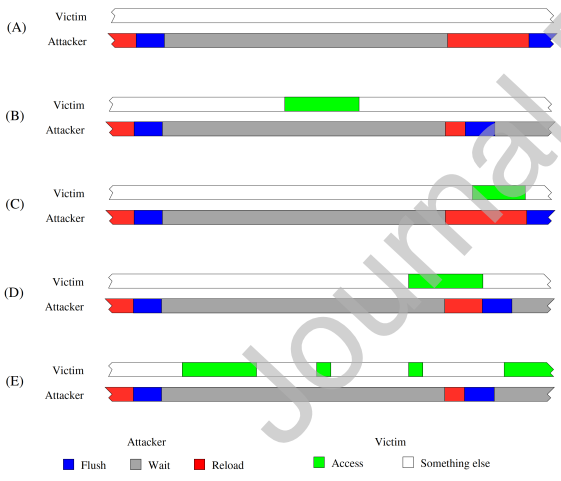


Figure 1: Timing of Flush+Reload. (A) No Victim Access (B)With Victim Access (C) Victim Access Overlap (D) Partial Overlap (E) Multiple Victim Accesses [8].

2.3. RSA – Exploitation by Flush+Reload Attack

RSA is a cryptographic cipher typically used for encryption and signing. It computes modular exponents in order to encrypt or sign data. There are many implementations of RSA distinguished upon the procedure used for computation of modular exponents. One of the computation procedure found in RSA is known as Square-and-Multiply. We skip math behind this procedure because cache side-channel attacks exploit implementation rather than math behind procedure. Therefore, we discuss the implementation details of Square-and-Multiply only. Algorithm 1 shows the part of the Square-and-multiply implementation of RSA. This code takes s (boolean array of secret bits), x (character of plain text) and m (value of modulus) as an input, and it returns result of $x^s \ mod \ m$ (modular exponent) as an output. To compute such exponent, this code executes Square (Line 4), Multiply (Line 7) and Barrett (Lines 5 and 8) operations in each iteration of loop (Line 3). The sequence of these operations depend on a bit value of s (Lines 3-10) in each iteration, i.e, computing sequence of Square-Barrett-Multiply-Barrett on HIGH bit of s but computing only Square-Barrett on LOW bit of s. Intuitively, each of these operations accesses different memory locations during execution, which means accessing of specific memory location corresponds the activation of secret-dependent operations (Square, Multiply or Barrett). In the view of attacker who can extract the memory access traces via cache side-channel attacks, it can recover the secret bits by back tracing the memory accesses. For example, Figure 2 shows the sequence of operations made by Square-and-Multiply implementation of RSA that are extracted through Flush+Reload attack and back traced bits of key. In this figure, red, purple and green colors represent the memory accesses by Square, Multiply and Barrett operations respectively. Additionally, the horizontal length of each color indicates multiple memory accesses taken per one activation of specific operation because attacker has targeted the loops in the each procedure. The consecutive Square-Barrett operations indicate skipping of Multiply-Barrett operation because the condition of "If" statement in Algorithm 1 (Line 6) is false, thereby recovering the LOW secret-bit against first Square-Barrett operation.

Algorithm 1 Square-and-Multiply implementation of RSA

```
1:  procedure SQU–MUL-EXP ( s[], x, m )
2:      r ← 1;
3:      for i from n − 1 downto 0 do
4:          r ← SQUARE(r);                    ▷ square operation
5:          r ← Mod(r, m);                    ▷ barrett operation
6:          if s[i] == 1 then
7:              r ← Multiply(r, x);           ▷ multiply operation
8:              r ← Mod(r, m);                ▷ barrett operation
9:          end if
10:     end for
11: end procedure
```

3. Flush+Prefetch –Proposed Countermeasure

Flush+Reload is a category of cache-based SCAs that has demonstrated the extraction of cryptographic key by monitoring cache accesses generated by RSA decryption process. The Flush+Prefetch countermeasure takes benefit from two limitations of Flush+Reload type of attacks. The first limitation is the fact that such attacks cannot identify the source (thread) that has fetched data in cache line. This limitation can be elaborated in a situation where the attacker thread is targeting a cache line that

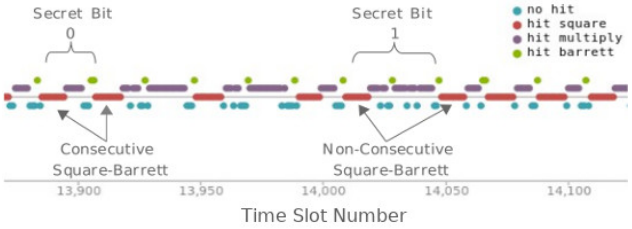


Figure 2: Graphical representation of memory accesses of RSA extracted via cache side-channel.

is shared between multiple concurrent threads. The attacker, in his reload phase, cannot distinguish whether the cache accesses are being generated by the concerned thread (victim) or any other concurrent thread. Thus, cache accesses generated by unconcerned threads, i.e., other than victim, are *noise* from the attacker's perspective. We refer such noise as *Positive* noise since it has a positive effect on the execution time of victim thread due to increased cache hits.

The second limitation is the fact that such attacks cannot detect multiple operations on a particular cache line. Exploiting this limitation enables the countermeasure to *hide* or *misrepresent* the information related to the exact cache accesses of the victim thread. This limitation can be elaborated in a situation where the attacker evicts a particular cache line in his first phase (i.e., eviction) and then waits for the victim to access that cache line. During this wait phase of the attacker, if the victim or some other concurrent thread evicts the concerned cache line *after* being used by the victim and immediately *before* the reload phase of the attacker, it will result in a cache *miss* for the attacker, which was otherwise supposed to be a cache *hit* from the attacker's perspective. This increases the attacker's likelihood of missing cache accesses by victim. Thus, the pattern shown in Figure 2 that is required by the attacker is either hidden or misrepresented. We refer such noise as *Negative* noise since it has a negative effect on the execution time of victim thread due to the possibility of increased cache miss for the victim. Eviction of the concerned cache lines by an independent concurrent thread can potentially effect the hit rate of both victim and attacker threads.

The term *Noise* refers to the extraneous memory/cache operations that are used to obfuscate memory access information of application leaking via cache side-channel attacks. Depending on the type of memory/cache operations, we introduced two types of noises, i.e. (1) *Positive noise* that is generated using memory read operations and (2) *Negative noise* that is generated using cache flush operations. Given that the application having temporal locality and is under attack, the word positive and negative with noise indicate the decreasing and increasing effect on the execution time of the application under attack respectively, which is discussed in detail in Section 3.1 and 3.2.

The Flush+Prefetch countermeasure against Flush+ Reload attack on Square-and-Multiply implementation of RSA cryptosystem uses both positive and negative noise and their combination to preserve confidentiality. Flush+Prefetch creates independent concurrent threads for positive and negative noise that share victim thread's address space. We have selected the way of adding noise using independent threads rather than integrating countermeasure (prefech and flush instruction) in RSA code because of performance reasons. Integration of prefetch and flush instructions in RSA will greatly degrade the performance. This is because we have to fetch all security critical instructions before use and have to add in critical path of RSA program (similarly as in single path programming). However, in our proposed countermeasure, the prefetch and flush instructions are executing through independent threads generated by the RSA process and not in the critical path of RSA program.

In contrast to previous countermeasures, Flush+Prefetch benefits in the following ways. First, the integration of noise threads does not raise incompatibility with super pages because it is designed to use standard widely available interfaces of computer systems without modification, i.e., multithreading environment, multithreading libraries, and instructions like prefetch and clflush. Second, Flush+Prefetch does not suggest disabling of SMT feature to mitigate first-level-cache attacks because it adds noise in all cache levels by prefetching or flushing locations from all the cache levels. Lastly, Flush+Prefetch mitigates the cache attacks even though the data-deduplication is allowed in the computer system. Data-deduplication enables the sharing of memory among the process and makes it easy to launch the cache attacks. Our countermeasure obfuscates the secret information extracted through the cache side-channel. Therefore, the information leaking via cache attacks using shared or unshared memory becomes irrelevant to our countermeasure.

3.1. Positive Noise

The positive noise thread uses *prefetch* instructions to fetch memory locations in the cache to add extraneous memory accesses. These extraneous memory accesses are independent of the security critical information and therefore obfuscate the memory access information. This noise thread is designed to execute concurrently with the victim thread. Therefore, different cases are possible based on memory accesses made by positive noise thread relative to victim thread. Figure 3 shows these possibilities.

Case-A shows that positive noise thread has only generated the memory access during wait phase of the attacker. The attacker, in this case, cannot distinguish the source of access (positive noise or victim) and takes memory access as generated by the victim. This confusion results in loss of temporal pattern of cache accesses (discussed in Section 2.2). Hence, the victim achieves confidentially through obscurity. Case-A is further elaborated in Figure 4 with real execution trace. Figure 4 shows the cache access pattern captured by attacker in the presence of positive noise, which is introduced in *Square* operations of RSA (Square-and-Multiply implementation). In Figure 4, square hits in the highlighted area are due to the prefetching of positive noise thread instead of square operations being performed by the victim thread. During reload phase of the attacker, the positive noise thread makes it difficult for the attacker thread to distinguish between actual square operations performed by the

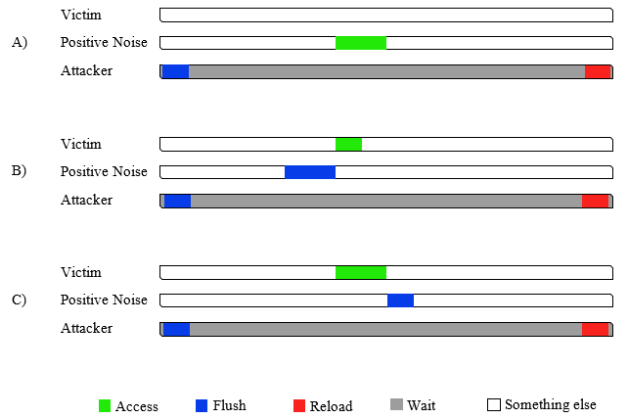


Figure 3: Timing information of Flush+Prefetch: Different cases of positive noise.

victim thread and the prefetching operations performed by the positive noise thread.

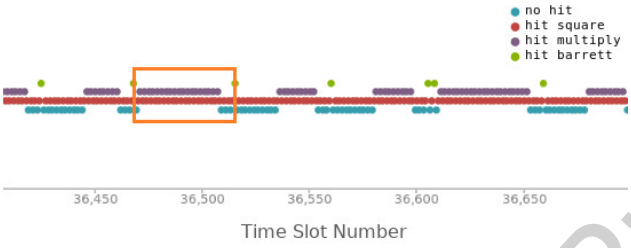


Figure 4: Cache access pattern: Prefetching by positive noise thread in Square & Multiply loops.

Case-B in Figure 3 shows that both positive noise thread and victim thread have generated the memory access during wait phase of the attacker. During reload phase, the attacker deduces correct information that victim has actually generated a memory access. In this case, positive noise thread is acting only as a prefetch for the victim thread. This particular case does not help improving confidentiality. However, given that the victim having temporal locality, the latency of some memory access made by victim is reduced because of prefetching victim's data in cache and this results in faster execution of victims program under attack.

Case-C in Figure 3 is similar to Case-B except the positive noise thread is executed after the victim thread. This situation contributes neither in achieving confidentiality nor in improving the execution time of victim thread. Same as in Case-B, both positive noise and victim threads have accessed the memory during the wait phase of attacker. Therefore, during reload phase, the attacker deduces correct information that victim has actually generated a memory access. Also, in the wait phase, the positive noise thread is executed *after* the victim thread that does not help victim to prefetch data from main memory. Thus, the execution time remains the same as without positive noise thread.

Given that the application is under attack, cases A,B and C occur multiple times in the presence of positive noise. The overall effect of these cases results in obfuscating the memory access trace leaked via cache side channel attack and improved execution time of application under attack.

3.2. *Negative Noise*

The negative noise thread uses *cflush* instruction to evict the cache lines targeted by the attacker. The negative noise thread executes concurrently with other threads similar to the positive noise thread. Therefore, different cases are possible based on instant of execution of negative noise thread relative to other threads. Figure 5 shows these possibilities.

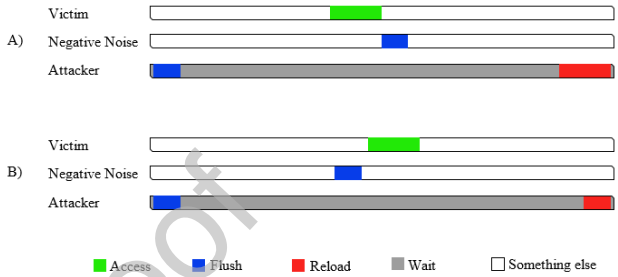


Figure 5: Timing information of Flush+Prefetch: Different cases of negative noise.

Case-A in Figure 5 shows that, during the wait phase of attacker, the negative noise thread is executed *after* the victim thread. In this case, the negative noise thread evicts the shared cache line that victim thread had cached earlier and used. The attacker, in its reload phase, would still register a cache miss as it will not find information cached by the victim. Thus, the attacker deduces incorrect information about the victim's access pattern as victim has actually generated a memory access, which is evicted by the negative noise thread immediately after use. This causes the victim's access invisible to attacker.

Case-B in Figure 5 shows that the negative noise thread is executed *before* the victim's thread. This situation does not contribute in hiding victim thread's access pattern from the attacker in reload phase. The attacker, in its reload phase, would register a cache hit that is correct information about victim's access pattern. This helps the attacker to capture some of the accesses by victim as shown in highlighted area in Figure 6 and the negative noise will not be useful. Figure 6 shows negative noise being introduced in the Barrett operation of RSA.

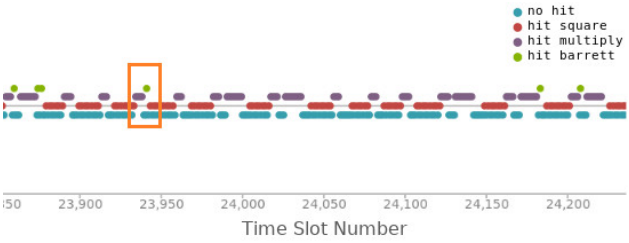


Figure 6: Cache access pattern: Eviction by negative noise thread in Barrett loop.

Another expected case between victim and noise threads is that both request the same memory location at the same time.

The bus arbiter makes such concurrent memory accesses of both threads in a sequence of which order depends on the arbiter policy. Therefore, this case ends up becoming either Case B or C of Figure 3 or either Case A or B of Figure 5.

3.3. *Design Cases for Square-and-Multiply implementation of RSA*

Square-and-Multiply implementation of RSA is vulnerable because of the existence of relation between key bits and accesses form instruction-cache (or I-cache), particularly related to the Square procedure in Algorithm 1, as discussed in Section 2.3.

There are three vulnerable procedures in Algorithm 1 that relate to the secret key , i.e, Square, Multiply and Barrett, . From Attacker's perspective, the activation pattern of Square and Multiply procedures reveal information related to the *length of secret key* and the *number of HIGH bits* in the key, respectively. Activation of Barrett procedure indirectly relates to the key because it is activated after every individual activation of Square and Multiply procedures. For example, if the length of key is 1024 bits with 505 HIGH bits in it, then the activation of Square, Multiply and Barrett procedures will be 1024, 505 and 1529 (= 1024 + 505) times, respectively.

The rationale for Flush+Prefetch technique is motivated from the fact that blind noise injection will not preserve confidentiality even when the relationship between key bits and access of instruction-cache lines is known. We argue that, it might seem as if injecting noise in I-cache lines that belong to Square procedure of Algorithm 1 will preserve the confidentiality of private key, but it is not sufficient. To support this argument, we consider an example of execution of three threads namely; an Attacker thread with Flush+Reload technique, a Victim thread containing Square-and-Multiply implementation of RSA, and a Positive Noise thread targeting cache lines related to Square procedure. Figure 4 shows the resulting pattern captured by the attacker. The attacker was able to capture the "hit square" and "hit multiply", which represent cache accesses for instructions of Square and Multiply procedures, respectively, as shown in the highlighted zone. Since Square-and-Multiply algorithm is not designed to execute any two procedures in parallel, therefore, the actual cache accesses for execution of Square procedure can be easily separated by ignoring the "hit square" that occur in parallel to "hit multiply" and secret key information can still be retrieved. Nevertheless, the attacker needs to modify Flush+Reload technique to perform this separation of access type. It is imperative, however, to add noise in an intelligent manner in cache lines belonging to all procedures that somehow relate to the key to ensure confidentiality. Along with the security, performance is the another aspect that need to be taken care while the selection of noises for memory locations. Negative noise negatively impacts the performance if added on the memory locations that are most frequently accessed by RSA within the short period of time. Whereas, positive noise positively impacts the performance either the memory locations is frequently or least frequently accessed by RSA. To find which type of noise should be used for interested memory location, it is required to analyze the impact on performance experimentally by individually adding each noises on interested memory locations. Based on this experiment, we have shown that memory locations belonging to barrett procedure of RSA can be used with negative noise in section 4.1.

The Flush+Prefetch technique presents two cases of noise injection in multiple cache lines of vulnerable procedures to ensure confidentiality with minimum performance degradation. In the first case, Flush+Prefetch technique introduces positive noise in cache lines belonging to all vulnerable procedures (Square, Multiply, and Barrett) simultaneously, as illustrated in Figure 7. From the perspective of confidentiality, such noise makes it difficult for the attacker to extract an otherwise obvious access pattern. Another advantage of such noise injection is that it decreases cache misses for the victim and therefore improves its execution time as discussed in Case-B of Section 3.1.

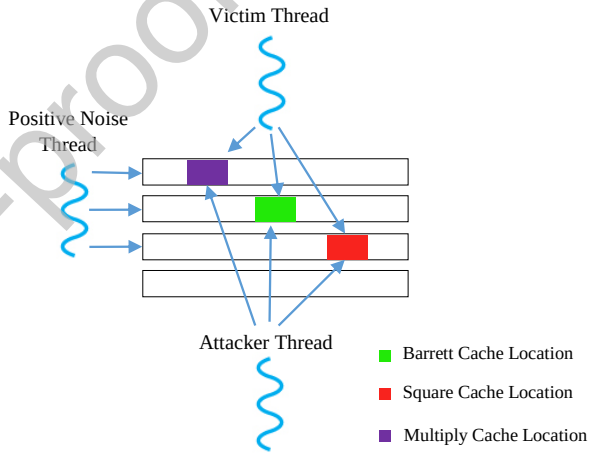


Figure 7: Design Case 1: Positive noise in Square, Multiply & Barrett loops.

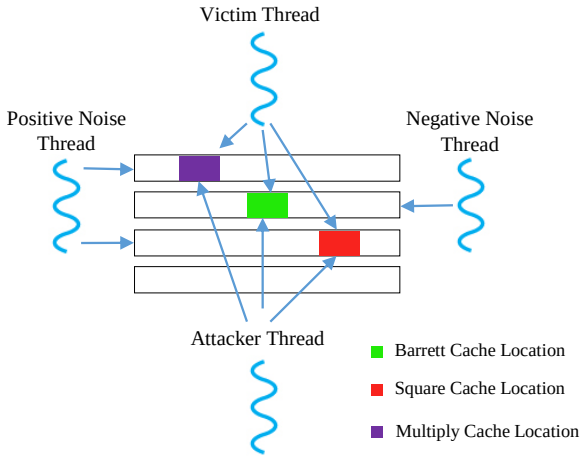


Figure 8: Design Case 2: Positive noise in Square & Multiply loops, negative noise in Barrett loop.

In the second case, Flush+Prefetch technique proposes to in-

ject mixed noise using both positive and negative noise threads. In this case, the negative noise is injected in cache lines belonging to Barrett procedure and the positive noise is injected in cache lines belonging to Square and Multiply procedures as illustrated in Figure 8. From the perspective of confidentiality, mixing negative noise with positive noise on selected procedures makes it very difficult for the attacker to extract any specific access pattern. A potential drawback of negative noise injection could be the increased performance penalty as the victim will have increased cache misses. Flush+Prefetch technique addresses this issue by injecting negative noise in cache lines belonging to Barrett procedure *only* as the number of accesses generated for cache lines belonging to Barrett procedure are much less than Square and Multiply procedures.

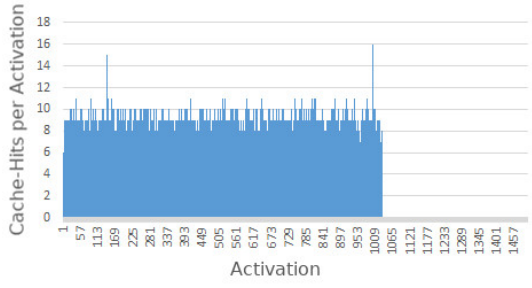
4. Experimental Evaluation

All experiments were performed on workstation with an Intel Xeon E5-2643 CPU, operating at a fixed clock of 3.40 Ghz, and 256 GB of RAM. The operating system used was Ubuntu 16.04 LTS. The size of L1 cache is 32KB, L2 cache is 256KB, and L3 cache is 10MB. The size of each cache line is 64 bytes. On average the number of instructions per cache line are 18 for the case of RSA program used in our experimental setup. In RSA code, total number of cache lines belong to Square, Multiply and Barrett procedures are 8, 9 and 20 respectively.

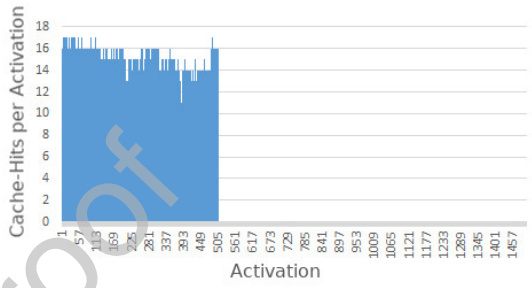
We had enabled super-pages, data-deduplication and simultaneous-multithreading and created four threads namely; Victim, Attacker, Negative Noise, and Positive Noise. Victim was a small SSL web-server developed by axTLS Embedded SSL using Square-and-Multiply implementation of RSA [36]. This was configured to use the key for cryptographic operation of 1024 bits in length comprising 505 HIGH bits. In these experiments, Attacker thread monitors the web-server process using Flush+Reload technique [37] after every 500 cycles (or time slot). The threshold used in attack to differentiate between cache hit and miss is set to 80 cycles. Positive and negative noise threads are developed using $prefetch$ and $clflush$ instructions, respectively, and these are set to high priority than the other threads. We performed 100, 000 iterations and monitored confidentiality and execution time of web application using cryptographic operations with and without the proposed noise for each loop of vulnerable procedures.

Another important configuration of noise threads is to set the memory addresses, which are pointed by *prefetch* or *clflush* instructions. Memory addresses used by noise threads typically divided into two parts, i.e. offset (static part) and base addresses (dynamic part). To obtain offsets, we used assembly file generated using *objdump* command. As base address of victim application changes on each execution depending on the frame availability in main memory, we used *dlsym* function while execution of victim application to get base address. Then these obtained base and offsets addresses are added to calculate the exact memory addresses, which are used by *prefetch* or *clflush* instructions in the noise threads.

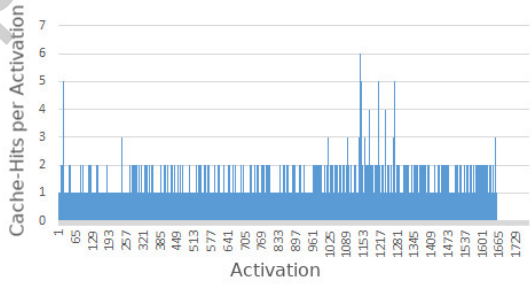
To evaluate the security of proposed solution, we have developed the baseline by obtaining the results of attacker without



(a) Reference for Square



(b) Reference for Multiply

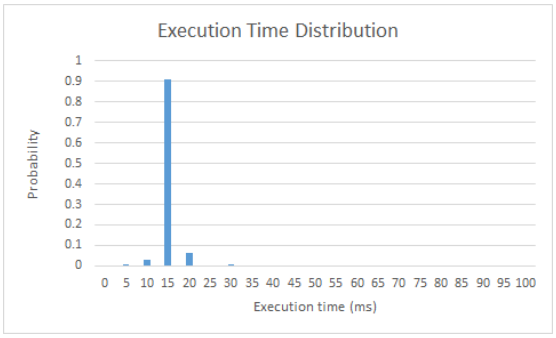


(c) Reference for Barrett

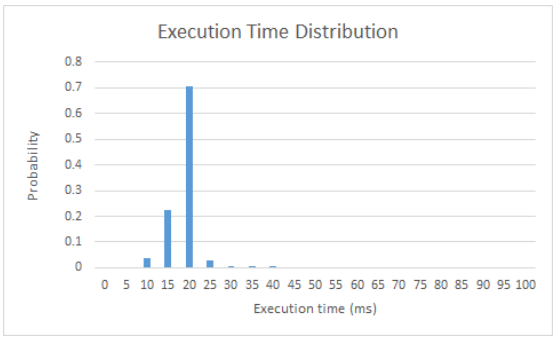
Figure 9: Activation pattern without noise, taken as a reference for confidentiality of (a) Square procedure (b) Multiply procedure (c) Barrett procedure.

noise. Results are shown in Figure 9a,9b and 9c. Figure 9a shows that square-activations are 1024 which is equal to the key length (i.e. 1024 bits). Figure 9b shows that multiply-activations are 505 which is equal to the number of HIGH bits in key (i.e. 505 bits). Figure 9c shows that barrett-activations are about 1529 which is equal to the sum of total length (i.e. 1024 bits) and the number of HIGH bits (i.e. 505 bits) in the key. Later on, we show that such relation vanishes in the presence of the positive or negative noises. Figure 9a also illustrates that the attacker has captured multiple cache hits (i.e. usually $8 - 11$) on each square-activation. This is because the attacker are designed to target the loop within the square procedure.

To evaluate the performance, execution times of web sever in the presence and absence of attacker was computed for 100, 000 turns and its mean taken as lower and upper bound of execution times, respectively, for cryptographic operation in the rest of experiments. Figure 10a shows the execution time of crypto-



(a)



(b)

Figure 10: Execution time distribution of (a) victim thread alone (b) victim thread in the presence of attacker thread.

graphic operation when only victim is operating. We observe that 90% runs of cryptographic operation complete in 15 ms. The mean of distribution is 15.13 ms, which is taken as a lower bound of execution time for cryptographic operation. Deviation in execution time is because of the fact that underlying processor architecture has components such as pipelines, branch prediction and other speculative components [38]. We then executed victim thread in the presence of attacker thread and observed that most of the runs lie at 20 ms which is roughly 70% of our total 100,000 runs and only 22.4% of total runs lie on 15 ms. Hence the execution time of cryptographic operation is increased by 24% in the presence of attacker thread as shown in Figure 10b. The mean of distribution in this case is 18.64 ms, which is taken as an upper bound of execution time for cryptographic operation. This increase in execution time is because attacker evicts the cache lines, which increases the cache misses for victim thread as discussed in Section 2.2. Figure 11 shows a graphical representation of cache hits and misses captured by the attacker thread for instructions belonging to loop in *Square*, *Multiply* and *Barrett* procedures in the absence of any noise. We refer them as *square loop*, *multiply loop*, and *barrett loop* in the results.

4.1. *Confidentiality and Performance Evaluation under Noise Injection in Single Procedure*

This section presents activation pattern and effect on execution time of web application by injecting either positive or negative noise alone in cache lines that belong to each procedure individually. These results prepare the ground for selection of

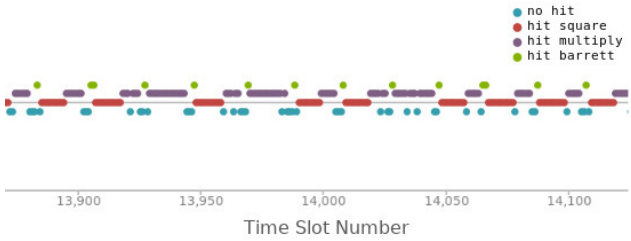
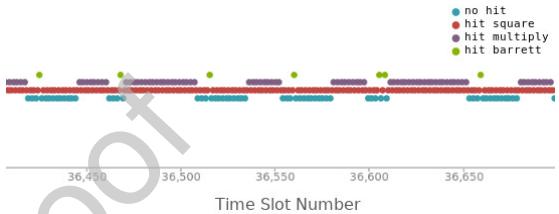
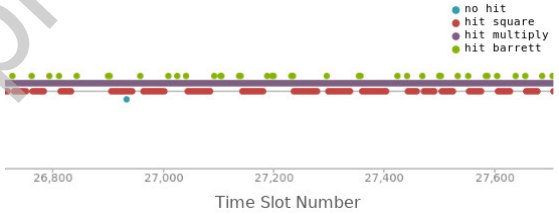


Figure 11: Graphical representation of cache access pattern without noise.

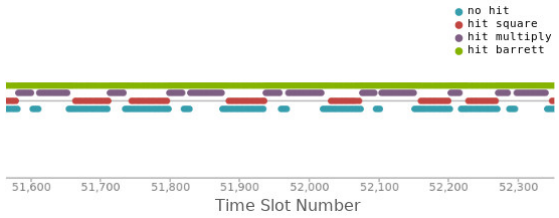
positive or negative noise for procedures in our design cases discussed in Section 3.3.



(a)



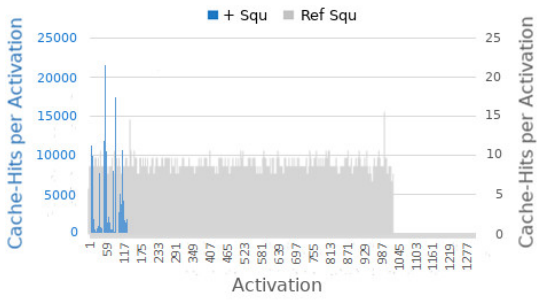
(b)



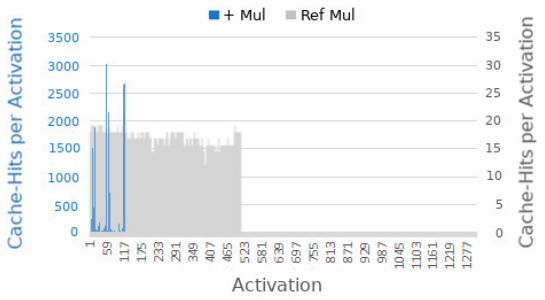
(c)

Figure 12: Graphical representation of cache hits and misses with positive noise in (a) Square procedure (b) Multiply procedure (c) Barrett procedure.

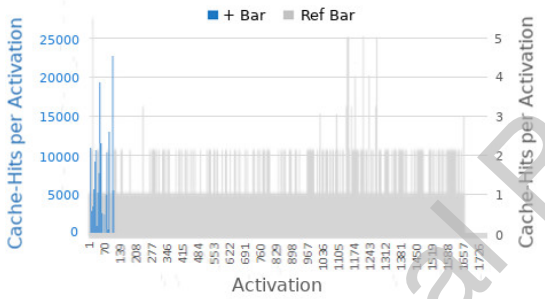
In this case, we have observed the activation pattern and execution time of web application in the presence of positive noise thread targeting cache lines related to all three loops of interest, i.e., Square, Multiply, and Barrett loops. Results given in Figures 12a-12c are related to positive noise injection in all loops. Considering square loop for instance, Figure 12a shows that inactive intervals between consecutive activations of square procedure are now filled by cache accesses generated by the positive noise thread. Therefore, from attacker's perspective, the square instruction is perceived as being *continuously accessed* by victim thread. Results in Figure 13a shows number of cache hits per each instance of square-activation, which is different than reference pattern given in Figure 9a (i.e., our reference pattern). First difference is that the total number



(a)

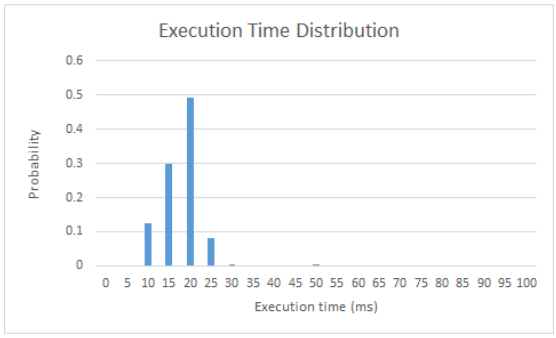


(b)

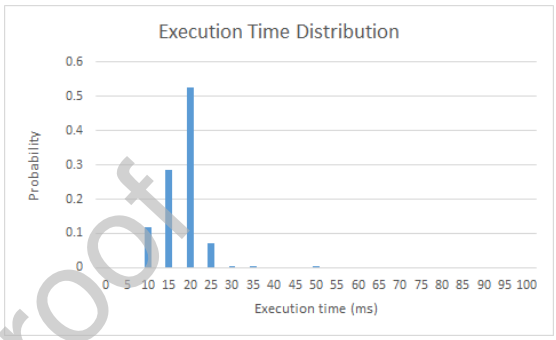


(c)

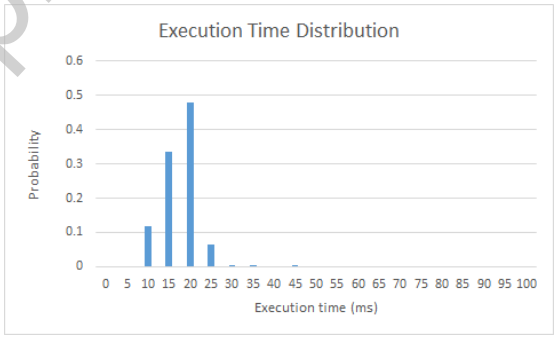
Figure 13: Activation patterns in the presence of positive noise in (a) Square procedure (b) Multiply procedure (c) Barrett procedure.



(a)



(b)



(c)

Figure 14: Execution time distribution of victim's process with attacker and positive noise in (a) Square procedure (b) Multiply procedure (c) Barrett procedure.

of square-activations captured by the attacker are 125, which is 87.8% less than the square-activation-pattern given in Figure 9a. This is because the attacker can no more distinguishably identify each and every activation and captures much less instances. The attacker can only capture an accumulated set of instances and registers them as a *single* instance. Consequently, the second difference is that the number of cache hits are enormously increased in each (accumulated single) instance of square-activation captured by the attacker. Based on these results, if the attacker considers the total number of activation instances, it cannot extract secret key due to much less captured instances. Similarly, the number of cache hits in each such captured instance is out of proportion with the reference pattern given in Figure 9a (cache hits shown on y-axis for each instance).

Results shown in Figure 14a present the execution time and its mean is given in Table 1. This concludes that the mean execution time is 0.9 ms less than the mean execution time of cryptographic operation in the presence of attacker thread alone. Execution time is reduced because the positive noise has increased cache hits for the victim thread, as discussed in the Case-B of Section 3.1.

Similar to the square loop, we have repeated the experiment and injected positive noise in cache lines that belong to multiply loop. In this case too, results shown in Figure 12b, demonstrate the absence of inactive intervals between consecutive multiply-activations. In this case, as shown in Figure 13b, the attacker was able to capture 117 multiply-activations, which is 77% less than the reference pattern given in Figure 9b. Figure 13b shows that number of cache hits for each instance is pretty random and usually greater than 100, which is different than the *usual* number of cache hits (i.e., 12 – 17 hits) in reference pattern (Figure 9b). Figure 14b shows the execution distribution and its mean is

given in Table 1. The mean execution time in this case is 1 ms less than the execution time of cryptographic operation in the presence of attacker thread alone due to instruction-prefetching behaviour of positive noise.

Lastly, we have injected positive noise in Barrett loop. Results in this case, as shown in Figures 12c, 13c and 14c show similar behaviour as discussed before. Figure 12c show that the inactive intervals between consecutive barrett-activations are filled with positive noise accesses. Figure 13c shows that the attacker was able to capture 108 barrett-activations, which is 93% less than the reference pattern. Number of cache hits reported in Figure 14c also showed similar behavior as in case of square and multiply loops.

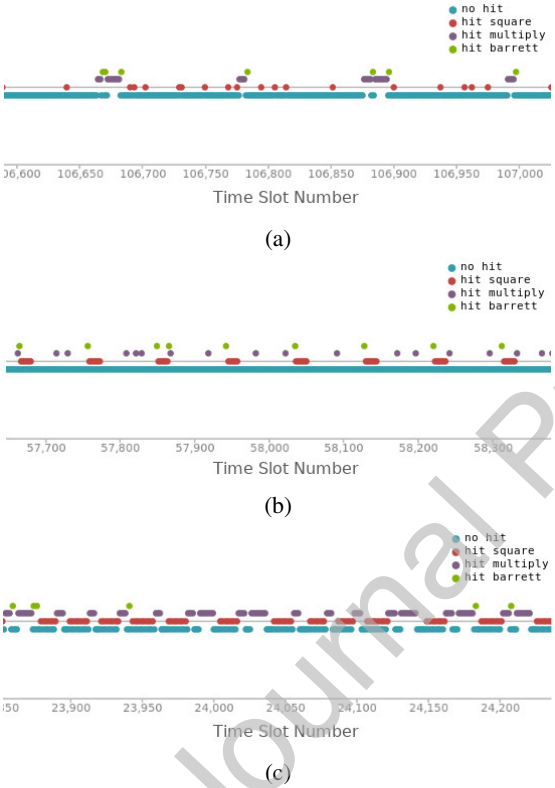


Figure 15: Graphical representation of cache hits and misses with negative noise in (a) Square procedure (b) Multiply procedure (c) Barrett procedure.

Inversely, we also present our observations on the activation pattern and execution time of web application in the presence of negative noise thread targeting cache lines related to Square, Multiply, and Barrett loops separately. Results given in Figures 15a-15c are related to negative noise injection in all loops. Results in Figure 15a show that attacker is unable to capture most of the accesses for cache lines that belong to square procedure because of cache-line eviction by the negative noise thread. Figure 15a also shows that some of the activations are still captured by the attacker because of the reason discussed in Case B of Section 3.2. Results in Figure 16a present execution-time distribution that shows the significant execution time degradation of 11.8 ms greater than the execution time in the presence of

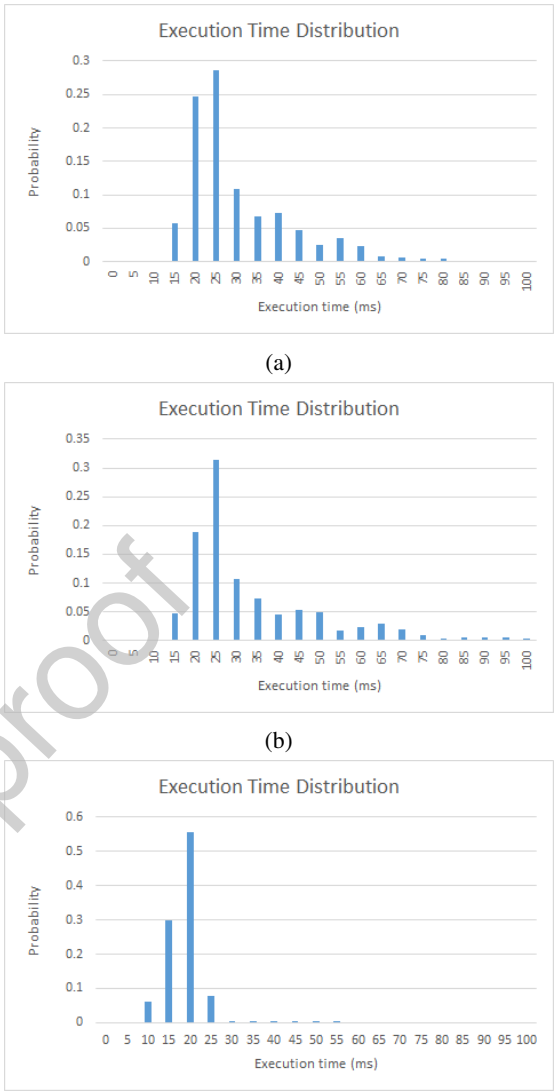


Figure 16: Execution time distribution of victim's process with attacker and negative noise in (a) Square procedure (b) Multiply procedure (c) Barrett procedure.

attacker alone. This is because of the negative noise evicts the cache lines that web application is accessing frequently. Although negative noise does create confusion for the attacker, this case is not practically viable because of the significant execution time degradation.

In the second step, we inject negative noise in multiply loop. In this case, the results in Figure 15b show that the attacker is unable to capture most of the accesses for cache lines that belong to Multiply procedure. However, just like the case of square loop, some of the activations are still captured by the attacker as shown in Figure 15b for similar reasons. As far as the execution time distribution is concerned, results shown in Figure 16b depict a significant shift of execution time over higher values and results in high mean value. Its mean is given in Table 1, which is 14.5 ms greater than the execution time in the presence of attacker alone.

Lastly, we inject negative noise in barrett loop and observe the effects. Results in Figure 15c show that the attacker is now unable to capture most of the barrett-activations. This result is different than the noise injection in square and multiply loops in a way that the attacker, in most of the attempts, cannot capture even one cache access per barret-activation. This is because the cache accesses made by each barrett-activation is much less than square and multiply activations. Figure 15c shows that very few of the barrett-activations, however, are still captured by the attacker because of the reason discussed in Case B of Section 3.2. Figure 17 shows that the barrett-activations being captured are 77, which is 5% of the reference pattern shown in Figure 9c. As far as the execution time distribution is concerned, results shown in Figure 16c depict minor shift of execution time towards higher values. Its mean is given in Table 1, which concludes that the mean execution time is about 1.6 ms greater than the execution time in the presence of attacker alone.

4.2. All-Positive and Mix-Noise Cases

Based on the results obtained for noise injection in individual instructions loops, we introduce two specific design cases of Flush+Prefetch. These design cases are developed with the aim of achieving confidentiality with minimum possible performance overhead. In the following, we discuss these cases one-by-one.

4.2.1. Design Case-1: Concurrent Positive Noise in all Instructions (Square, Multiply, and Barrett loops)

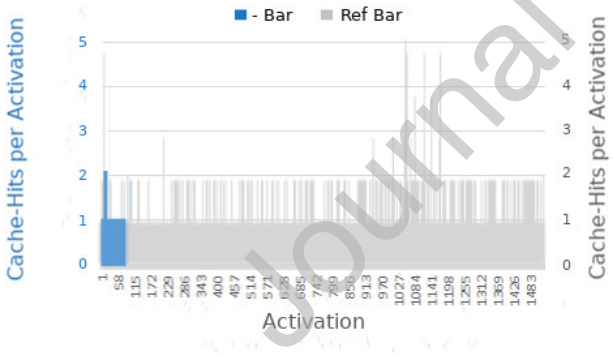


Figure 17: Barrett pattern in presence of negative noise.

In this design case, positive noise is injected in all vulnerable cache lines related to Square, Multiply, and Barrett procedures simultaneously. Results in Figure 19b show the attacker captured cache access pattern, which depicts that the victim had executed all procedures in parallel. In fact, the positive noise fills the inactive intervals between consecutive activations of all the procedures.

Results in Figures 18a, 18b and 18c show the number of cache hits per each vulnerable procedure activation captured by the attacker respectively. These results are different from the reference activation patterns in two ways. First, there is lack of consistency in the resulting pattern as compared to reference pattern for all procedures. Second, the captured activation instances of all procedures are much less as compared to reference activation patterns.

This is because of the fact that, earlier, the attacker used to determine the completion of an instance activation based on inactive intervals between any two consecutive activation instances. These inactive intervals, in this case, are now filled by the positive noise. As a result, the attacker perceives a continuity of activation instance without finding any inactive interval and thus, cannot determine the completion of an activation instance. Figures 18a, 18b and 18c also show that now the attacker is capable of capturing the square, multiply and barrett activations only up to 195, 74 and 242, which are 19%, 14.5% and 15.8% of the reference patterns respectively. These results reveal that the attacker will capture a random number of activation instances. Moreover, these captured instances will have no correlation with actual key bits anymore.

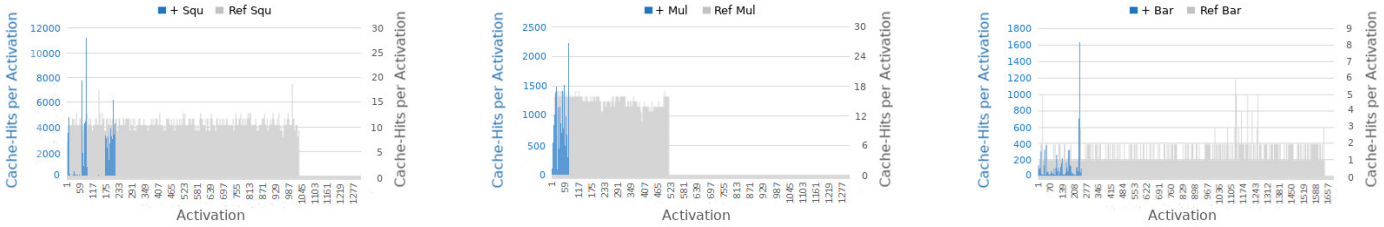
Results in Figure 20b and Table 1 show the execution time distribution and its mean of Design Case-1 respectively. Thanks to the positive noise, victim does not experience any performance overhead in terms of execution time as compared to performance in the presence of attacker alone. Mean execution time in this design case is 1.9 ms less than the execution time of cryptographic process in the presence of attacker alone. This results in 10.2% improvement in execution time of web server application (victim process) while under Flush+Reload attack (given in Figure 20a).

4.2.2. Design Case-2: Concurrent Positive Noise in Square & Multiply Loops and Negative Noise in Barrett Loop

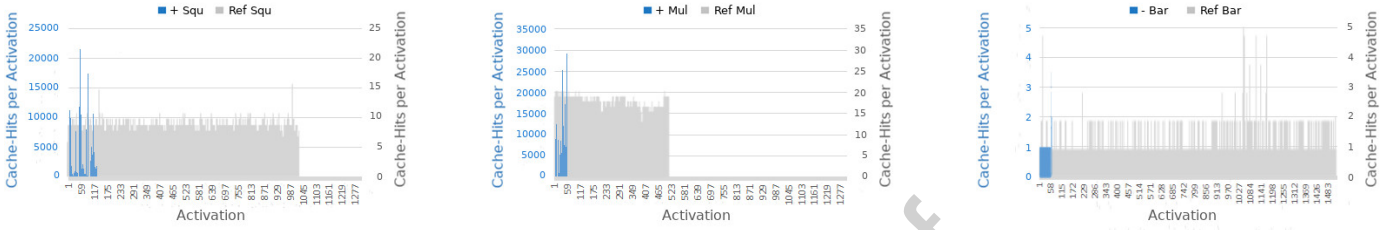
In this design case, results are obtained by injecting two types of noise, simultaneously, i.e., positive noise in Square and Multiply loops and negative noise in Barrett loop. Figure 19c shows the cache access pattern in this design case. These results show two effects as compared to results given in Figure 19a (case with no noise injection). First effect is the elimination of inactive intervals between consecutive square and multiply-activations due to positive noise as discussed in the Design Case-1 as well. Second effect is the cache misses between consecutive barrett-activations due to negative noise injection as discussed in Case-A of Section 3.2.

Results in Figures 18d and 18e show number of cache hits per square and multiply activation captured by the attacker, respectively. These results are similar to the ones discussed in Figures 18a and 18b. Positive noise injection fills the inactive intervals, which leads to a cache access pattern with much less activation instances and much higher hit rates for the attacker. Figures 18d and 18e show that square and multiply activations captured by attacker are only 229 and 59, which are 22.3% and 11.6% of reference square and multiply activation patterns respectively.

Results in Figure 18f show the number of cache hits per barrett-activation captured by the attacker. We obtained a significant reduction in the number of captured activation instances

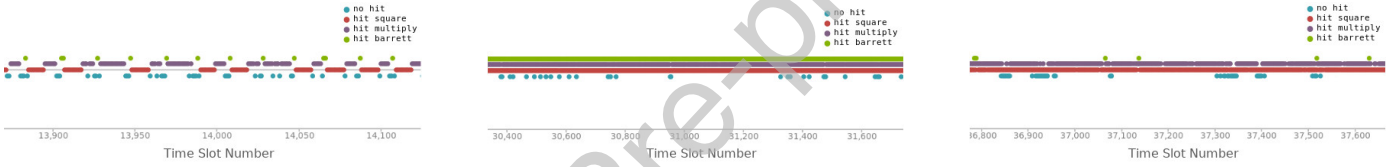


(a) Square pattern of Design Case 1 (b) Multiply pattern of Design Case 1 (c) Barrett of Design Case 1



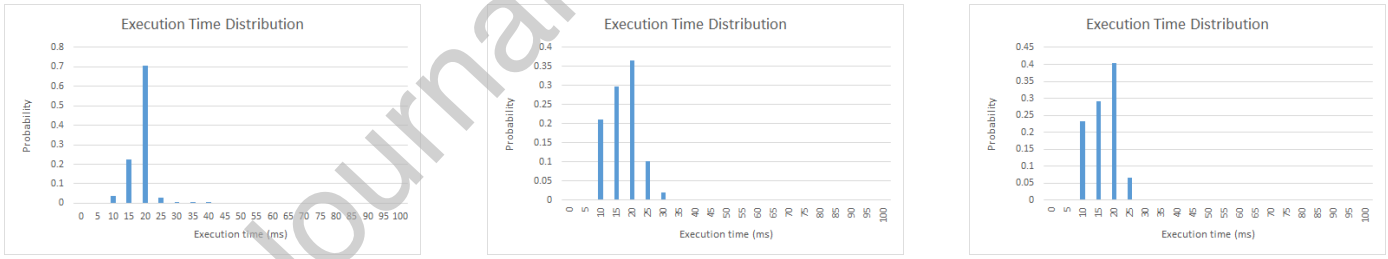
(d) Square pattern of Design Case 2 (e) Multiply pattern of Design Case 2 (f) Barrett pattern of Design Case 2

Figure 18: Activation patterns for Design Cases Design Case-1 (a)-(c) and Design Case-2 (d)-(f) +Squ: Positive noise in Square loop, +Mul: Positive noise in Multiply loop, +Bar: Positive noise in Barrett loop, and -Bar: Negative noise in barrett loop



(a) Reference (b) Design Case 1 (c) Design Case 2

Figure 19: Graphical representation of cache hits and misses of reference and design cases.



(a) Reference (b) Design Case 1 (c) Design Case 2

Figure 20: Execution time distributions of reference and design cases

compared to the reference pattern. This is due to the negative noise injection in vulnerable cache lines related to Barrett procedure. Barrett-activations captured by attacker are 77, as shown in Figure 18f, which are about 95% less as compared to the number of activations in reference pattern.

Results in Figures 18d, 18e and 18f reveal that the attacker will capture an even more random number of activation instances for all instructions compared to Design Case-1. Moreover, these captured instances will have even lesser correlation with actual key bits as well. Calculated mixing of positive and negative noise enhances the confidentiality aspect of the cryptographic operations with a significant margin in this design case.

Results in Figure 20c show the execution time distribution and Table 1 shows its mean for Design Case-2. Mean of this distribution is 17.1 ms, which indicates 8% improvement in execution time of web server application (victim process) when under attack (given in Figure 20a).

5. Security and Performance Comparison

To the best of our knowledge, we have found that single path programming based countermeasure [24] outperforms as compared to previous application level countermeasures [10, 35]. Therefore, we have compared the security and

Table 1: Comparison of effect of positive and negative noises on execution time (Legends are such as +Squ: Positive noise in Square loop, +Mul: Positive noise in Multiply loop, +Bar: Positive noise in Barrett loop, -Squ: Negative noise in square loop, -Mul: Negative noise in multiply loop and -Bar: Negative noise in barrett loop).

	Mean (*ms*)
Victim alone (Reference Design)	15.1
Victim+Attacker	18.6
+Squ	17.7
+Mul	17.6
+Bar	17.8
−Squ	30.4
−Mul	33.1
−Bar	18.3
Design Case 1 (+Squ,+Mul,+Bar)	16.7
Design Case 2 (+Squ,+Mul,−Bar)	17.1

Table 2: Security Comparison

Countermeasures	NCD
Single path programming	0.746
Flush+Prefetch	0.762

performance of Flush+Prefetch with single path programming based countermeasure.

For security and performance comparison, we have converted the Square-and-Multiply implementation of RSA given in web-server by axTLS [36] into the single path as shown in Algorithm 2. All the inputs and outputs of Algorithm 2 are same as compared to unmodified implementation given in Algorithm 1. The main difference of Algorithm 2 as compared to original Algorithm 1 is that it operates the same sequence of operations Square-Barrett-Multiply-Barrett (Line 5 to 8) whether the bit value of secret is LOW or HIGH. For correct operation, after executing operations, code updates the variable depending on the key value (Line 9).

For security comparison of single-path programming with our countermeasure, we have computed the Normalized Compression Distance (NCD) between original and retrieved keys form each countermeasure. NCD is used to measure the similarity between two objects and it ranges from 0 to 1 where 0 means identical and 1 means completely different [39]. In our case, we use this to show the effectiveness of obfuscation introduced by each countermeasure by showing the dissimilarity introduced in side channel information in the presence of countermeasure, this is same method used in [40] to prove the effectiveness of its countermeasure. Table 2 shows the NCD for both countermeasures. This shows that Flush+Prefetch countermeasure achieves same security as single-path programming.

We have evaluated the performance of both Flush+Prefetch and single path programming based countermeasure on the same computing setup used for security evaluation. Additionally, libpfm-4.10.0 library is used to measure the execution time of both countermeasures. We have taken the measurements of 100,000 runs of both systems. Then mean of these measurements are calculated and compared. Table 3 shows the execution overhead of both countermeasures. We observed that execution time of RSA modified to single path programming overhead as compared to unmodified RSA is 72%. In case of Prefetch+Flush countermeasure, execution time overhead as compared to to unmodified RSA is only 10%. Flush+Prefetch outperforms 62% as compared to RSA modified to single path programming. The execution time of single path approach is significantly larger because it executes costly multiply operation in each iteration of loop regardless of the secret bit.

Table 3: Execution Time Comparison with Unmodified Web Server

	Execution Time Overhead Versus Unmodified Web Server
Single path programming	×1.72
Flush+Prefetch	×1.10

Algorithm 2 Single Path implementation of RSA

```
1:  procedure SQU–MUL-EXP ( s[], x, m )
2:      r ← 1;
3:      r2 ← r;
4:      for i from n − 1 downto 0 do
5:          r ← SQUARE(r);                ▷ square operation
6:          r ← Mod(r, m);                ▷ barrett operation
7:          r2 ← Multiply(r, x);          ▷ multiply operation
8:          r2 ← Mod(r, m);               ▷ barrett operation
9:          r ← (s[i] == 1) ? r2 : r ;
10:     end for
11: end procedure
```

6. Discussion

6.1. Synchronization of Threads

Synchronization between threads can enhance the security. However, synchronization introduces a performance overhead. Our work has shown that synchronization between victim and noise thread is not necessary till assumption of fair scheduler remains valid. Linux OS scheduler schedules each thread fairly, which directly ensures the addition of noise. Our results are justifying that the access-trace of vulnerable cache addresses becomes unintelligible to the attacker while relying on OS scheduler and without synchronization.

6.2. Generalization of Technique

As we discussed in related work, the countermeasure that prefetches the AES tables [10] incorporated the *prefetch* instruction within the application's execution flow. This requires

modifying application radically to introduce *prefetch* instruction within the application. Hence, a generalization of this countermeasure is difficult for each application. But in our countermeasure, the *prefetch* instruction is independent of the application execution flow. Only memory addresses targeted by the attacker are required. These addresses can be figured out by seeing assembly files. These addresses are taken as input to noise threads and launched independently.

6.3. Secret Information Leakage form Data Cache

Flush+Prefetch countermeasure can obfuscate the leakage of secret information from data cache as well. This is because Flush+Prefetch countermeasure obfuscates accesses of interested cache lines only based on memory addresses, regardless of the fact that these addresses are mapped to instruction cache or data cache. For example in case of AES, memory accesses of T-table elements, which are cached in data cache, depend on the secret key. So prefetch and flush threads are provided with memory addresses that are mapped to T-tables. This will obfuscate the cache access footprint of AES algorithm.

Leakage of secret information depends on the algorithm. For example in case of RSA, execution of square and multiply instructions depends on key bits but operates on the same data. Therefore, instruction cache access will reveal key bits and data will be accessed in both cases whether the key is HIGH or LOW, so sequence of data cache accesses does not leak secret information. In contrast to RSA, AES secret key is leaked in exactly opposite to that of RSA. In case of AES, instruction access is same and data access varies depending on the secret key. Therefore, data cache accesses is interesting for the attacker. Our countermeasure takes memory addresses irrelevant to whatever the memory contain instruction or data. To counter data leakage using proposed countermeasure, the prefetch and flush threads target the memory addresses that are mapped to security critical data (such as T-table in AES) in data cache.

6.4. Mitigating Prime+Probe Attack using Flush+Prefetch Countermeasure

Prime+Probe attack also has three phases same as Flush+Reload attack but the way of probing the cache is different. In first phase of Prime+Probe attack, attacker initializes the interested cache line state by filling the interested cache lines. Then in second phase, attacker waits for predefined time interval while that the victim executes and utilizes the cache. In third phase, attacker reads again the cache lines and measures the read latency. If victim has loaded some cache lines in second phase, it will evict the cache lines and results in increase in read access latency by attacker in third phase, indicating the access of cache line by victim. In our countermeasure, prefetch thread loads the cache line similarly as victim, so prefetch thread also causes eviction and results in increase in access latency measured by attacker in third phase. Moreover, flush thread in our countermeasure also causes eviction of the cache lines and results in increase in access latency measured by attacker in third phase. Access pattern obtained using Prime+Probe will include the cache line accesses generated by victim, prefetch and flush threads, hence, the access pattern is obfuscated.

6.5. Core Utilization

For instance, it looks like that proposed countermeasure will occupy additional cores and performance overhead will be large. However, today SMT feature of CPU is disabled in servers because of the collocation of multiple threads can raise cache-based side-channel attacks. Disabling SMT feature underutilized the CPU resources because of data and control dependencies between instructions in the program. Therefore, instructions of noise threads, which are independent to RSA instructions (as in our countermeasure), can be fetched in single core along with RSA instructions and results in use of underutilized resources to mitigate the cache attacks in case of SMT enabled CPU.

We have measured the instructions per cycle (IPC) of *axtls* application to evaluate the underutilization of CPU. CPU having the top speed of 4.0 IPC while executing the *axtls* application can execute on average 2.92 instructions per cycle, which means that the axtls application under utilizing CPU of about 27% (= $(1 - 2.92/4)$). The under-utilization of CPU can be used to execute instructions for security. In our case, we have limited the utilization of CPU by noise instructions (*prefetch* and *clflush*) by up to 1 IPC, which is within the range of measured underutilization of CPU, and observes sufficient addition of noise for security as shown by the results in Figures 19b and 19c.

7. Conclusions and Future Work

This paper proposes a novel application-level countermeasure technique, called Flush+Prefetch, against the Flush+Reload category of cache-based side-channel attacks. The proposed countermeasure is easily deployable and works without the requirement of specialized hardware features or any profound changes to system-level software. Flush+Prefetch technique uses intelligent noise injection to improve confidentiality of the victim process, i.e., the applied cryptosystem. The countermeasure uses independent threads that consist of $prefecht$ and $cflush$ instructions to generate noise. As a result, the cache access pattern of victim application is encoded in such a way that the attacker, using Flush+Reload attack, cannot capture the actual cache access pattern. We have evaluated the confidentiality and performance of Flush+Prefetch countermeasure technique on Intel Xeon E5-2643 processor using $100,000$ cryptographic rounds of Square-and-Multiply implementation of RSA cryptosystem. Our experimental results show that the confidentiality of cache accesses made by RSA is preserved under Flush+Prefetch technique as the leakage of information is reduced to 22.3% only as compared to 96.7% bits recovery reported by Flush+Reload attack in a single decryption round. Results show that the leaked information is scattered and does not contain any specific pattern, i.e., either bit position or bit value, that can facilitate the establishment of secret key for RSA cryptosystem even if the attacker intends to do multiple iterations. Our results show that the performance, in terms of average execution time, is improved by 10.2% for best design case compared to the system under Flush+Reload attack.

Flush+Prefetch technique practically demonstrates that noise-based solutions are viable countermeasures and good candidates for *quick-patch* solution against precision attacks like Flush+Reload. The proposed countermeasure can be extended for other types of cache-based SCAs such as Prime+Probe in the future. Moreover, as a future work, we intend to integrate detection mechanisms with our proposed countermeasure in order to further improve performance.

References

References

[1] V. Millnert, J. Eker, E. Bini, Feedback for increased robustness of forwarding graphs in the cloud, Journal of Systems Architecture 80 (2017) 68 – 76. doi:https://doi.org/10.1016/j.sysarc.2017.09.005.
URL http://www.sciencedirect.com/science/article/pii/S138376211730070X

[2] J. Zhou, T. Wang, P. Cong, P. Lu, T. Wei, M. Chen, Cost and makespan-aware workflow scheduling in hybrid clouds, Journal of Systems Architecture (2019) 101631doi:https://doi.org/10.1016/j.sysarc.2019.08.004.
URL http://www.sciencedirect.com/science/article/pii/S1383762119302954

[3] G. Irazoqui, T. Eisenbarth, B. Sunar, S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES, SP '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 591–604. doi:10.1109/SP.2015.42.
URL http://dx.doi.org/10.1109/SP.2015.42

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, R. B. Lee, Last-Level Cache Side-Channel Attacks are Practical, 2015, pp. 605–622. doi:10.1109/SP.2015.43.

[5] Y. Zhang, A. Juels, A. Oprea, M. K. Reiter, HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis, 2011, pp. 313–328. doi:10.1109/SP.2011.31.

[6] Y. Zhang, A. Juels, M. K. Reiter, T. Ristenpart, Cross-Tenant Side-Channel Attacks in PaaS Clouds, CCS '14, ACM, New York, NY, USA, 2014, pp. 990–1003. doi:10.1145/2660267.2660356.
URL http://doi.acm.org/10.1145/2660267.2660356

[7] N. Eltayieb, R. Elhabob, A. Hassan, F. Li, An efficient attribute-based online/offline searchable encryption and its application in cloud-based reliable smart grid, Journal of Systems Architecture 98 (2019) 165 – 172. doi:https://doi.org/10.1016/j.sysarc.2019.07.005.
URL http://www.sciencedirect.com/science/article/pii/S1383762119300402

[8] Y. Yarom, K. Falkner, FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, USENIX Association, San Diego, CA, 2014, pp. 719–732.

[9] D. Boneh, Twenty Years of Attacks on the RSA Cryptosystem, Vol. 46(2), 1999, pp. 203–213.

[10] E. Brickell, G. Graunke, M. Neve, J.-P. Seifert, Software mitigations to hedge aes against cache-based software side channel vulnerabilities, jean-pierre.seifert@intel.com 13192 received 13 Feb 2006 (2006).
URL http://eprint.iacr.org/2006/052

[11] M. Godfrey, M. Zulkernine, A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud, 2013, pp. 163–170. doi:10.1109/CLOUD.2013.21.

[12] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, M. M. Swift, Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense), CCS '12, ACM, New York, NY, USA, 2012, pp. 281–292. doi:10.1145/2382196.2382228.
URL http://doi.acm.org/10.1145/2382196.2382228

[13] J. Shi, X. Song, H. Chen, B. Zang, Limiting Cache-based Side-channel in Multi-tenant Cloud Using Dynamic Page Coloring, DSNW '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 194–199. doi:10.1109/DSNW.2011.5958812.
URL http://dx.doi.org/10.1109/DSNW.2011.5958812

[14] T. Kim, M. Peinado, G. Mainar-Ruiz, STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud, Security'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 11–11.
URL http://dl.acm.org/citation.cfm?id=2362793.2362804

[15] M. M. Real, P. Wehner, V. Migliore, V. Lapotre, D. Ghringert, G. Gogniat, Dynamic spatially isolated secure zones for NoC-based many-core accelerators, 2016, pp. 1–6. doi:10.1109/ReCoSoC.2016.7533900.

[16] Z. Wang, R. B. Lee, A Novel Cache Architecture with Enhanced Performance and Security, MICRO 41, IEEE Computer Society, Washington, DC, USA, 2008, pp. 83–93. doi:10.1109/MICRO.2008.4771781.
URL http://dx.doi.org/10.1109/MICRO.2008.4771781

[17] F. Liu, R. B. Lee, Random Fill Cache Architecture, MICRO-47, IEEE Computer Society, Washington, DC, USA, 2014, pp. 203–215. doi:10.1109/MICRO.2014.28.
URL http://dx.doi.org/10.1109/MICRO.2014.28

[18] F. Liu, H. Wu, K. Mai, R. B. Lee, Newcache: Secure cache architecture thwarting cache side-channel attacks, IEEE Micro 36 (5) (2016) 8–16. doi:10.1109/MM.2016.85.

[19] Z. B. Aweke, T. M. Austin, Øzone: Efficient execution with zero timing leakage for modern microarchitectures, 2018, pp. 1123–1128. doi:10.23919/DATE.2018.8342179.
URL https://doi.org/10.23919/DATE.2018.8342179

[20] M. Taram, A. Venkat, D. M. Tullsen, Mobilizing the Micro-Ops : Exploiting Context Sensitive Decoding for Security and Energy Efficiency, 2018.

[21] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, X. Li, A Simple Cache Partitioning Approach in a Virtualized Environment, 2009, pp. 519–524. doi:10.1109/*ISPA*.2009.47.

[22] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, R. B. Lee, CATalyst: Defeating last-level cache side channel attacks in cloud computing, 2016, pp. 406–418. doi:10.1109/HPCA.2016.7446082.

[23] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, J. Emer, Dawg: A defense against cache timing attacks in speculative execution processors, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 974–987. doi:10.1109/MICRO.2018.00083.

[24] A. Rane, C. Lin, M. Tiwari, Raccoon: Closing digital side-channels through obfuscated execution, in: 24th USENIX Security Symposium (USENIX Security 15), USENIX Association, Washington, D.C., 2015, pp. 431–446.
URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[25] O. Aciiçmez, Yet Another MicroArchitectural Attack:: Exploiting I-Cache, CSAW '07, ACM, New York, NY, USA, 2007, pp. 11–18. doi:10.1145/1314466.1314469.
URL http://doi.acm.org/10.1145/1314466.1314469

[26] D. Gruss, C. Maurice, K. Wagner, S. Mangard, Flush+Flush: A Fast and Stealthy Cache Attack, DIMVA 2016, Springer-Verlag New York, Inc., New York, NY, USA, 2016, pp. 279–299. doi:10.1007/978-3-319-40667-1_14.

[27] D. Gullasch, E. Bangerter, S. Krenn, Cache Games – Bringing Access-Based Cache Attacks on AES to Practice, SP '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 490–505. doi:10.1109/SP.2011.22.
URL http://dx.doi.org/10.1109/SP.2011.22

[28] E. Tromer, D. A. Osvik, A. Shamir, Efficient Cache Attacks on AES, and Countermeasures, Vol. 23, 2010, pp. 37–71. doi:10.1007/s00145-009-9049-y.
URL http://dx.doi.org/10.1007/s00145-009-9049-y

[29] D. A. Osvik, A. Shamir, E. Tromer, Cache Attacks and Countermeasures: The Case of AES, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 1–20. doi:10.1007/11605805_1.
URL http://dx.doi.org/10.1007/11605805_1

[30] G. Irazoqui, M. S. Inci, T. Eisenbarth, B. Sunar, Wait a Minute! A fast, Cross-VM Attack on AES, Springer International Publishing, Cham, 2014, pp. 299–319. doi:10.1007/978-3-319-11379-1_15.
URL http://dx.doi.org/10.1007/978-3-319-11379-1_15

[31] M. Neve, J.-P. Seifert, Advances on Access-Driven Cache Attacks on AES, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 147–162. doi:10.1007/978-3-540-74462-7-11.
URL http://dx.doi.org/10.1007/978-3-540-74462-7-11

[32] L. Moritz, G. Daniel, S. Raphael, M. Clementine, M. Stefan, ARMageddon: Cache Attacks on Mobile Devices, Austin, TX, 2016.

[33] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Ham-

burg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, in: 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[34] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, in: 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, Baltimore, MD, 2018, pp. 973–990.
URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[35] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, M. Franz, Thwarting cache side-channel attacks through dynamic software diversity., in: NDSS, 2015, pp. 8–11.

[36] C. Rich, axtls embedded ssl.
URL http://axtls.sourceforge.net

[37] B. David, misc-cache-attacks.
URL https://github.com/polymorf/misc-cache-attacks/

[38] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools, Vol. 7, ACM, New York, NY, USA, 2008, pp. 36:1–36:53. doi:10.1145/1347375.1347389.
URL http://doi.acm.org/10.1145/1347375.1347389

[39] J. Mortensen, J. J. Wu, J. Furst, J. Rogers, D. Raicu, Effect of image linearization on normalized compression distance, in: International Conference on Signal Processing, Image Processing, and Pattern Recognition, Springer, 2009, pp. 106–116.

[40] A. Fell, T. Pham, S. K. Lam, Tad: Time side-channel attack defense of obfuscated source code, 2019. doi:10.1145/3287624.3287694.

Biography

Muhammad Asim Mukhtar is currently pursuing his PhD in Electrical Engineering from Information Technology University, Lahore, Pakistan. His research focuses on development of countermeasures against side channel attacks for embedded systems. He has MSc is in Electrical Engineering form The University of Lahore, Pakistan and BSc in Electrical Engineering from COMSATS, Lahore, Pakistan. Before undertaking doctoral studies in 2016, he worked as a Lecturer at The University of Lahore.

Maria Mushtaq received her MS degree in Computer Science from CIIT, Pakistan in 2014. She started her PhD in Information Security from University of South Brittany, France in 2016. Her research area focuses on cryptography, embedded system security, integration of security into cache-based side channels, identifying threat models in multi-core architectures and mainly proposing detection based and scheduling based mitigations to such attacks.

Muhammad Khurram Bhatti received his MS and PhD degrees in Computer Engineering from the University of Nice-Sophia Antipolis, France, in 2007 & 2011, respectively. He is the 2014 Marie-Curie Research Fellow (Postdoc) at KTH Royal Institute of Technology, Sweden. His areas of interest are Embedded system security, Hard Real-time and Mixed Criticality Systems, Contextaware Computing, Multicore Architectures, and Parallel Computing Systems. He is currently Director of Embedded Computing Lab and an Assistant Professor at Information Technology University (ITU), Pakistan.

Vianney Lapotre received his M.Sc. and his Ph.D. in Electrical and Computer Engineering from the University Bretagne Sud, France, in 2010 and 2013 respectively. In 2012 he spent six months as an invited researcher at the Ruhr-University of Bochum, Germany. From 2013 to 2014, he was a Postdoctoral at LIRMM, Montpellier, France. He was involved in the European Mont-Blanc project. He is currently associate professor at University Bretagne Sud, France. His research interests include hardware security, reconfigurable and self-adaptive multiprocessor architectures.

Guy Gogniat obtained his MSEE degree at the University of Paris Sud, Orsay, France, in 1995, and his PhD in electrical and computer engineering at the University of Nice-Sophia, Antipolis, France, in 1997. He is currently a Professor in electrical and computer engineering at the University of Bretagne-Sud, Lorient, France, where he has been since 1998. In 2005, he spent one year at the University of Massachusetts, Amherst, USA, as an invited researcher, where he worked on embedded system security using Reconfigurable technologies. His work focuses on the design of an d methods and tools for embedded systems. He also conducts research in reconfigurable and adaptive computing and embedded system security.