Exploring security of data on microprocessor hardware

M.Tech Thesis
Submitted in partial fulfillment of
the requirements for the degree of
Dual Degree Project
by

Meet Udeshi
(Roll No. 14D070007)

Supervisor:
Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

31 May 2019

Abstract

The current trends in computer architecture are increasingly focusing on sharing computing resources among multiple programs and users. Multiple programs can share a single core using simultaneous multi-threading which is widely supported by most processors and OSes. Virtual machine technology allows running multiple OS instances on the same processor. While the software and hardware of VMs or multi-threading OS is able to isolate illegal access of data to prevent exploits, it cannot prevent the leakage of sensitive data via side-channels which exist due to design flaws in shared hardware like caches, branch predictors, prefetchers. Attackers have successfully been able to extract encryption keys of various cryptographically secure algorithms like AES and RSA. These leakages are possible and viable because hardware design does not take care of the security against such side-channels. Moreover, software trojans can use these leakages to create a covert channel of communication unknown and undetectable by the OS and any software anti-viruses. Also, software exploits like return oriented programming and buffer overflow attacks can be thwarted more effectively with hardware solutions rather than software defenses. It has become increasingly necessary to consider data security as an important metric for hardware design.

This thesis first gives a summary of the various side-channel attacks using shared hardware which are present in literature. The summary gives a motivation for including hardware security as an important aspect of hardware design. An implementation of a reverse-engineering attack is described to extract parameters of the caches present in X86 cores. A denial-of-service attack on the prefetcher is described, followed by its implementation and test results. A hypothetical side-channel using the shared reorder buffer on SMT cores is presented.

Table of Contents

Abstract **i**

1 Introduction **1**

2 Side Channel Attacks **3**

2.1 Data dependent execution in encryption algorithms 3

2.2 Cache side channel 4

2.3 Prime+Probe 5

2.4 Flush+Reload 6

2.5 Reverse engineering cache parameters 7

2.5.1 Experimental setup 8

3 Covert Channel Attacks **11**

3.1 Cache channels on GPGPU 11

4 Mitigations against Cache side channels **13**

4.1 Partition-locked cache 13

4.2 Random permutation cache 14

4.3 Intentional Cache pollution 15

4.3.1 Disruptive Prefetching 15

4.3.2 Context sensitive decoding 16

5 Attack on Prefetcher **17**

5.1 Motivation 17

5.2 Implementation 18

5.2.1 Full attacker 19

5.2.2 Targeted attacker 19

5.3 Simulation and Results 21

5.3.1 Simulator setup 21

5.3.2 Results for benchmarks 21

5.3.3 Results for AES 22

6 Side-channel using Reorder Buffer **24**

7 Conclusion **25**

Chapter 1

Introduction

In recent years, it has become difficult to keep up with Moore's law using conventional transistor scaling. Computer architecture has shifted focus from optimizing single-thread performance to increasing throughput by running multiple threads and multiple programs simultaneously. The current trends are increasingly focusing on sharing computing resources among multiple programs and processes. Multi-thread and multi-core processors are commonplace in personal computers and mobile phones, even embedded devices. In cloud computing, technologies like virtual machines and virtual environments are allowing multiple different programs to share the same computing resources. These shared resources include all the structures inside cores and multi-core processors which can be accessed simultaneously by threads colocated on a single core, or even processes on two different cores. This poses a threat to the data security of many critical processes which run in such a shared context.

Attackers with the right knowledge and tools can leverage hardware implementation flaws in the design of these shared resources to extract data from a victim process via undetectable side-channels. Malicious trojans can use these shared resources to construct covert-channels to establish inter-process communication undetectable by the core or OS. With the rapid increase of need of powerful computation resources, GPUs have been extended to support general purpose computing. More recently, multiple processes are able to share the GPGPU resource and this opens up a new domain of security attacks which can be mounted on GPGPUs.

With the recent Meltdown (11) and Spectre (12) attacks capable of compromising any Intel core regardless of the OS, it is obvious that along with power and performance, design of computer architecture needs to consider data security as an important metric. Moreover, a lot of software based attacks like buffer overflow and return-oriented programming can be thwarted effectively using additional hardware structures. Hardware

support for security against software exploits is an efficient mitigation and should also be considered when designing processors.

A lot of side-channel attacks are based on caches due to their common accessibility between different programs. Cache accesses are abstracted away from the program hence OS-level access-restrictions do not apply on them. Caches also have a well-defined memory to cache-line mapping which is used by the attacker to infer memory addresses. The time-difference between a cache-hit versus a cache-miss is also noticeable enough to be detected by an executing program. These characteristics make the cache vulnerable to side-channel leakages (9). Cache designs which try to avoid any one of these characteristics lead to severe performance degradation. For example, if the memory to cache-line mapping is to be avoided, a fully-associative cache may be used instead of a direct-mapped or set-associative cache. But a fully-associative design limits the cache-size to a value much smaller than that desired by modern programs. In fact, the decision of moving from fully-associative caches to set-associative caches was done to make larger cache-sizes feasible on modern hardware, and that decision cannot be undone only for security measures without major impact on performance.

Cache designs like Newcache (8) try to achieve the same level of performance while also preventing side-channel leakage. There are other security methods which add enough noise to the cache to disrupt any side-channel. The Disruptive Prefetching (9) method utilises the function of prefetcher to generate random memory accesses to confuse an attacker while not interfering with program execution and performance. Another method introduces a new context-sensitive decoder (10) to mask legitimate memory access instructions with extra random accesses during instruction decode.

Side channels work best when only the targeted region of code of the victim is making memory accesses. While it is possible to prevent other programs and victim's irrelevant code from interfering, hardware which generates memory accesses like the prefetcher are difficult to stop. The thesis presents an attack on the prefetcher which tries to completely disable it from generating any memory accesses. This will help enhance the side channel to facilitate better and faster key extraction.

Chapter 2

Side Channel Attacks

Shared resources of the processor can leak information about the tasks being performed in it as shown in Fig. 2.1. This leaked information may be extracted by an attacker using various means. The attacker will try to use some form of measurement like cache hit/miss, time of execution, power consumption, EMI spikes to determine what part of code is running or what data is being processed (1). These kind of attacks have been proven to be effective on cryptography algorithms.

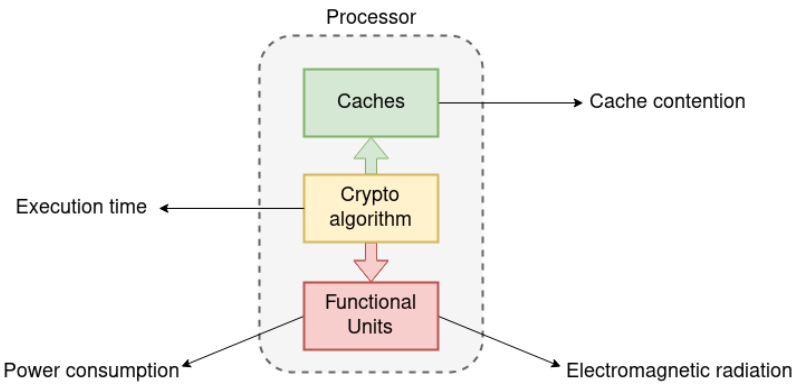

Figure 2.1: A number of side channels which are capable of leaking data.

2.1 Data dependent execution in encryption algorithms

Encryption standards like RSA, ECDSA, AES have been implemented in programs in a way which causes certain branches and memory access patterns to be dependent on the secret key. By using side channels to analyse which branch was taken or detect which memory address was loaded, it is possible to decode the secret key. For example, Listing 2.1 shows the key-dependent branch of fast exponentiation part of RSA. Fast exponentiation works by repeated squaring and multiplying when bit of the exponent is 1. In RSA, the

secret key is used as the exponent hence we get a bit-by-bit difference in executed code. When analysing power trace or measuring timing of the execution of this part of code, we can infer that higher power consumption and larger execution time occur when key bit is 1. This proves that there is information being leaked bit by bit.

Listing 2.1: Key-dependent branch of fast exponentiation used in RSA

```
while(key > 0) {
    e = key % 2;

    Square();
    Reduce();
    if(e == 1) {
        Multiply();
        Reduce();
    }

    key >>= 1;
}
```

In algorithms like AES and DES, P-box and S-box are used for fast permutation and substitution. They essentially store a mapped permutation or substitution for each key value. This means that during execution, AES algorithm will access various different blocks from S-box and P-box memory region depending on the key which is being used. If we can trace these memory accesses in some way, we can infer the secret key. Memory buses leak data about the address via EMI channel, and by analysing that we can get a trace of the memory access pattern. A better and more effective way of obtaining memory access patterns is by analysing the cache.

2.2 Cache side channel

All the threads running in a single core use the same L1 caches inside that core. Processes running on two different cores in a multi-core processor share the Last level cache. The data access patterns of a process leaves behind fingerprints in the cache. Because of set-associativity, if we can determine which cache line is being accessed by the process, we can determine the actual address which was accessed.

This is done without ever having to read the actual cache line, by causing contention on that cache line by an attacker process (2). When the attacker and victim are both trying to

use the same cache line, the attacker will get noticeable difference in execution time due to cache misses. There are various ways in which a cache side channel can be created.

2.3 Prime+Probe

The steps followed by Prime+Probe attack are as follows:

1. Attacker primes the cache line by loading his own data which .
2. Victim process runs and accesses memory mapped to same cache line, hence evicting attacker's data.
3. Attacker probes the cache line by reloading the same data, and looking for a cache hit/miss.

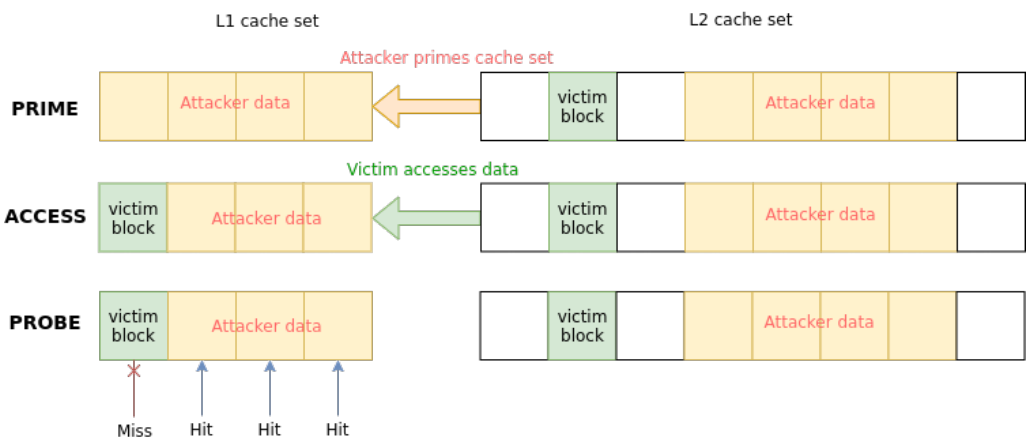

Figure 2.2: Example of a prime-probe attack on single L1 cache set. Miss in the PROBE step can be noticed by increased execution time

A miss in the probe step results in increased code execution time for the attacker, which it can easily measure by reading the Time Step Counter present in many modern cores. As shown in Fig. 2.2, attacker has to prime the entire cache set (all ways) for a successful attack. For analysing the victim's every memory access, attacker needs to prime the entire cache. This priming step leads to a lot of cache misses and can be tracked by event counters and trigger security exceptions when the cache misses reach an alarming amount. Moreover, in cases where attacker and victim are not colocated on the same core, such an attack would have to use a lower level of shared cache like LLC. The probing step requires LLCs to be fully inclusive else the victim will not evict attacker's data from the LLC and not lead to the required cache miss.

2.4 Flush+Reload

Flush+Reload is a side channel attack on caches which relies on the `clflush` instruction present in X86 ISA (and similar variants in other ISAs). Flush+Reload is able to work at a finer granularity than Prime+Probe. It is also able to successfully mount cross-core attacks via the LLC.

1. Attacker flushes a cache line using `clflush`.
2. Victim process runs and accesses memory hence loading the flushed block into cache.
3. Attacker reloads the same data, looking for a cache hit/miss.

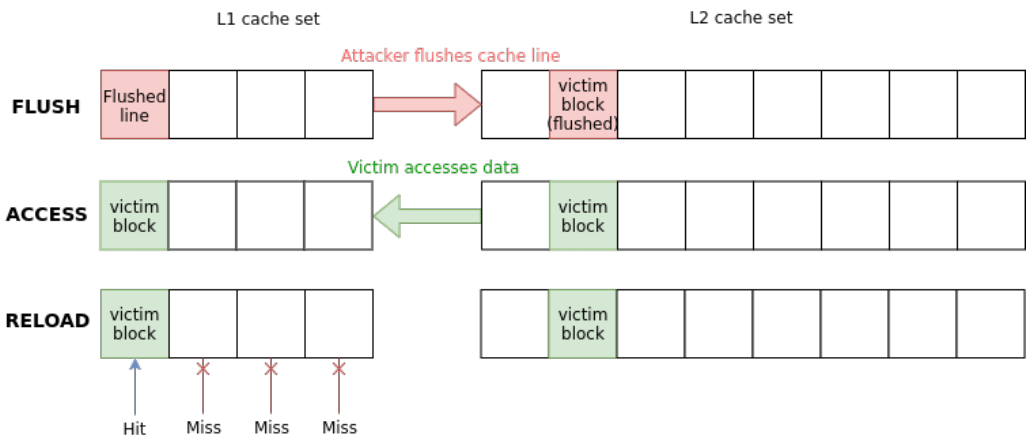

Figure 2.3: Example of a flush-reload attack on single L1 cache set. Hit in the RELOAD step can be noticed by decreased execution time

As seen in Fig. 2.3, the granularity of Flush+Reload is at cache line level rather than cache set level. This happens because the attacker tries to access the same data as the victim, instead of creating contention with other data mapping to the same cache set. Accessing same data is possible because majority of encryption algorithms are provided as system-wide shared libraries. Both the code and data regions of these libraries can be accessed by all processes. As opposed to Prime+Probe, this makes Flush+Reload a very practical and efficient attack. Flush+Reload is able to achieve greater granularity and accuracy due to it scanning for Cache Hit instead of Cache Miss.

Flush+Reload is also effective on LLCs because inclusivity will not affect `clflush` behaviour, hence attacker will get an LLC hit when the victim process accessed data. This opens up possibility of mounting a Cross-VM attack (3) like shown in Fig. 2.4

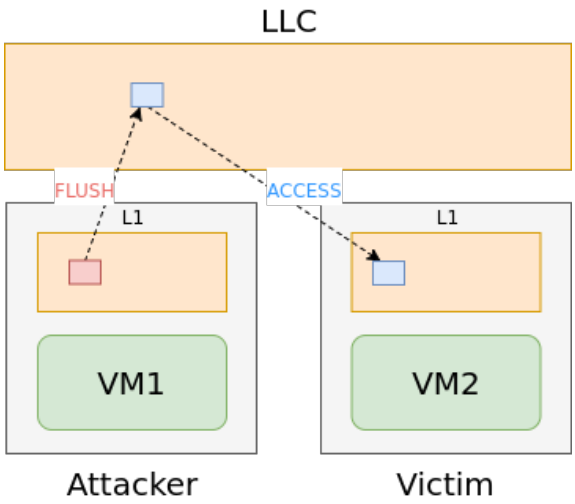

Figure 2.4: Flush+Reload via the LLC enables mounting Cross-VM attacks. This exploit is extremely significant in cloud computing environments

2.5 Reverse engineering cache parameters

This implementation of reverse engineering cache parameters is based on (5). In that paper, Wong et al show how using a stride access pattern over an array to trigger a predictable number of cache-misses. By measuring latency of stride access, we can get an idea of the number of cache misses.

For a given array size, we need to feed in the stride pattern into the array i.e. `array[i]` should contain location of `array[i+STRIDE]`. We can create such an array pattern offline before starting timing measurements. In this way, we can do a linked-list like traversal of the array without needing to calculate next stride location online. Listing 2.2 shows how one can create the array with a stride pattern.

Listing 2.2: Offline formation of array with stride access pattern

```
size_t* array; // malloced beforehand
size_t t;

for (int i=0; i<array_size; i++) {
    t = i + STRIDE:
    if(t >= array_size) t %= STRIDE;
    array[i] = (size_t)array + sizeof(size_t)*t;
}
```

For measuring timing of the array access, `rdtsc` instruction is used to get a reading of the Time Step Counter before and after accessing the array. The difference is plot versus

array size. Listing 2.3 shows how to traverse the array using the stride access data stored in it. The `next_ptr` variable stores pointer to next element to access. It is dereferenced and the loaded data is again stored into `next_ptr` for the next iteration.

Listing 2.3: Timing measurement of stride access over the entire array

```
long start = __rdtsc();
size_t* next_ptr = &array[0];
for(int i=0; i<MAX_ITERS; i++) {
    next_ptr = *((size_t**)next_ptr);
}
long time = __rdtsc() - start;
```

Fig. 2.5 shows a plot of latency vs array size. The latency plot stays constant initially until an array size which fills up the whole cache. Once that happens, some lines in the cache start getting evicted and we see a steep rise in latency. After any rise, the latency stays constant for the line size of the cache. This is obvious because any access in the same cache line will incur same total latency as there will only be one cache miss. This latency rise occurs once for each cache set, because as long as there are new cache sets to evict, there will be misses. The latency increase stops when one whole way of the cache is replaced once by the array access. The starting point of latency increase gives us cache size. The step width gives us line size. Number of steps gives number of sets, but that is hard to clearly determine when noise is present in measurements. Thus we determine way size by looking at the point where the latency plot flattens out again. Then sets = $\frac{\text{waysize}}{\text{linesize}}$.

2.5.1 Experimental setup

For all cases, stride of 64B was used.

One set of simulations was done using gem5 simple CPU and configurable cache sizes. This was done for testing out the algorithm. Fig. 2.5 was plot for L1 data cache of 1KB size, 2-way, 64B line size. Fig. 2.6 was plot for L1 data cache of 16KB size, 4-way, 64B line size.

The same algorithm was run on Intel Skylake i5-6500 processor with L1 cache of 32KB size, 8-way, 64B line size. The latency plot is shown in Fig. 2.7. As is seen, there is some amount of noise due to Out-of-Order processing and other programs interfering with the execution of the latency measurements. Despite the noise, we can clearly make out the steps, beginning of the latency increase, and way size. This gives us every parameter required for the cache.

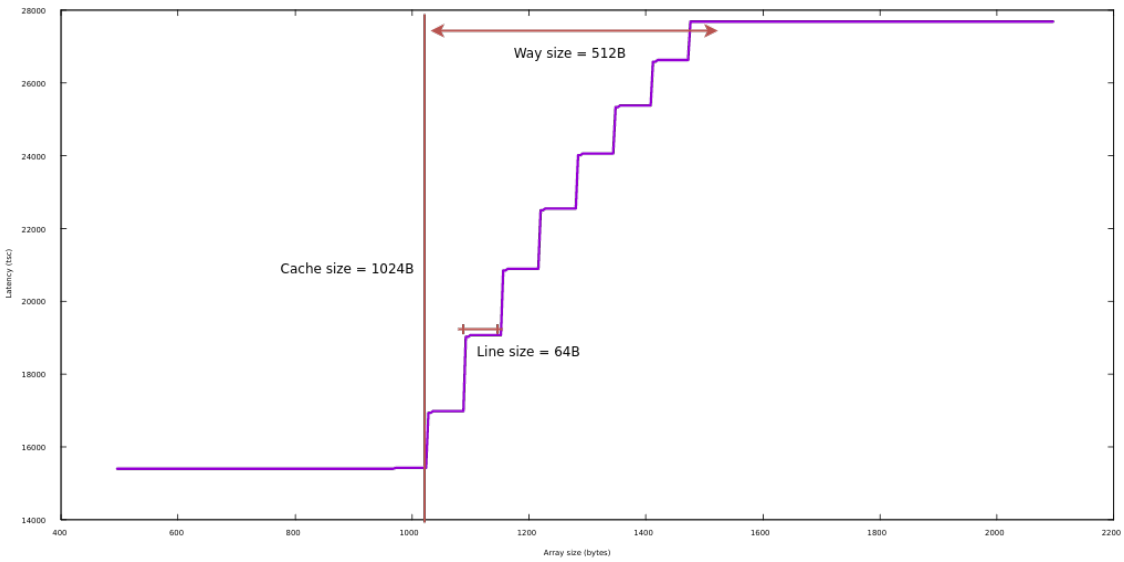

Figure 2.5: Latency vs. Array size plot for a 1kB 2-way cache with 64B cache line.

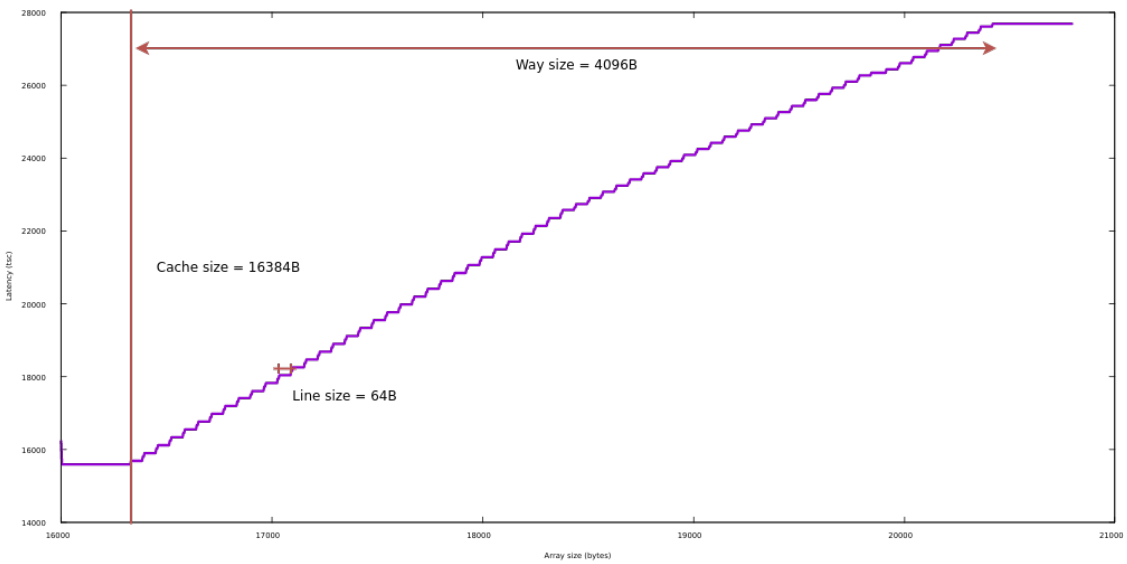

Figure 2.6: Latency vs. Array size plot for a 16kB 4-way cache with 64B cache line.

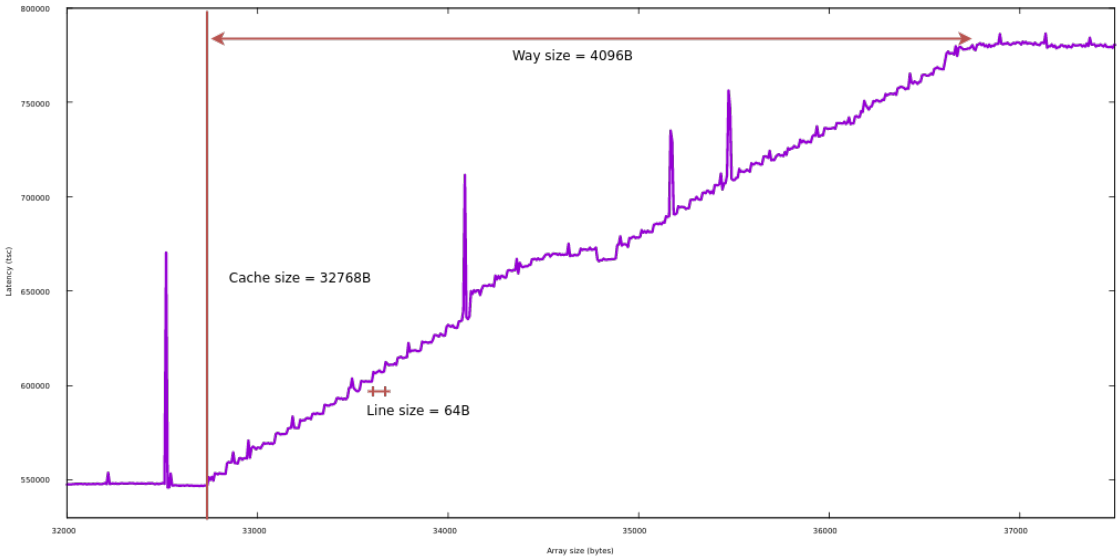

Figure 2.7: Latency vs. Array size plot for a 32kB 8-way cache with 64B cache line. Run on Intel Skylake i5-6500

Chapter 3

Covert Channel Attacks

3.1 Cache channels on GPGPU

GPGPUs have a massively parallel architecture which allows for SIMT workloads to efficiently run. Apart from graphics and display use-cases, GPGPUs are being used for parallel computation using frameworks like CUDA and OpenCL. GPGPUs in cloud services are specifically designed for such computational use-cases. Nvidia GPGPUs have recently started to support concurrent kernel execution at SM level, which allows multiple programs to simultaneously use the GPGPU resource. In this shared context, one must look at side channels which can be exploited.

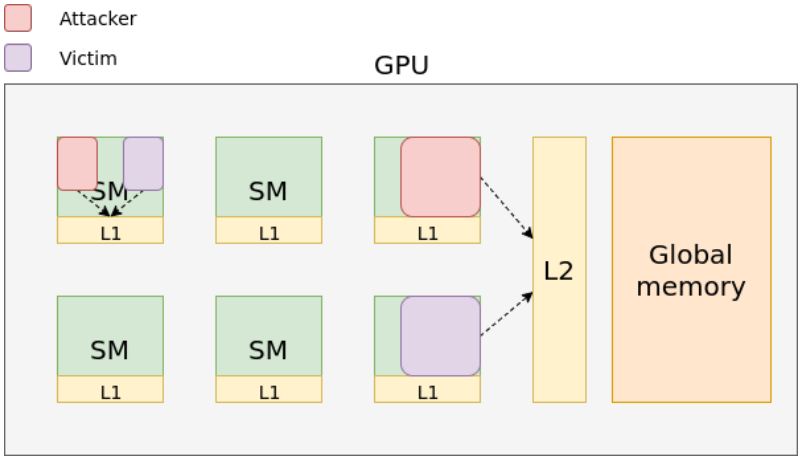

Figure 3.1: GPGPU memory layout. Attacker and Victim colocation allows using L1 and L2 caches as side channels.

The structure of GPGPU memory layout is shown in Fig. 3.2. Every SM contains a private L1 cache, and all SMs share an L2 cache. The Global memory contains multiple types of memory division like Constant memory, Texture memory etc. Concurrent kernel execution allows co-location of different kernels on same SM. Due to resource constraints,

kernels could also run on two different SMs simultaneously. The first case allows attacker to use L1 cache as side channel, and in the second case attacker has to use L2 cache.

Mounting a side channel attack on AES is possible on GPGPU because of existing implementations of AES for GPGPU. However, there are not many cases where encryption algorithms are run on GPGPUs. So these side channels are used as covert channels instead. Covert channels use the same methods as side channel but they are used to set up communication between two malicious programs. Such covert channels can be useful to leak data to third parties without the OS or hardware detecting malicious behaviour.

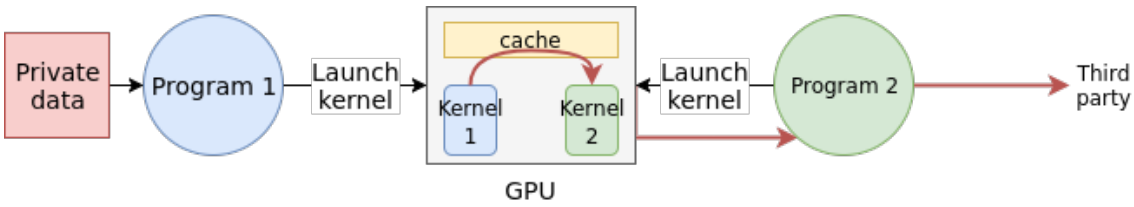

Figure 3.2: Structure of covert channel via GPGPU.

Naghibijouybari et al in (4) achieve communication speed of over 4Mbps using a combination of L1 cache contention and SFU contention as covert channels on multiple Nvidia GPGPU architectures. They have used the inherent parallelism in GPGPUs to multiply the speed of the created covert channels by opening parallel communication channels on each SM.

A critical part of their attack is reverse engineering various parameters of GPGPU architecture. To use caches as a side channel, we need to know all parameters of the cache structure. We also need to know of the warp scheduling policy to control colocation of two different kernels on same SM.

Chapter 4

Mitigations against Cache side channels

Software based mitigations against cache side channels involve changing the implementation of each encryption algorithm to avoid leaking data. But that is only possible for a specific set of known attacks, and it is unavoidable for any software to not leave some kind of fingerprint in the shared resources.

A proper solution involves changing hardware design of caches so that one process doesn't affect other processes via its cache accesses in a predictable way.

4.1 Partition-locked cache

Cache partitioning is a naive way of isolating processes from interfering with each other's cache accesses. A partitioned cache will let only one process access a single partition at a time (6). If we partition the cache statically, it is equivalent to having a private cache for every thread on the core. This either leads to huge power and area usage or high drop in performance.

Wang et al have proposed a dynamically partitioned cache in (7). As seen in Fig. 4.1, they add extra bits to every cache line to determine whether line is "Locked" and "ID" of thread which locked it. A modified cache replacement policy takes into account these bits when replacing any line. This ensures that locked lines can only be replaced by the process that locked them. Hence, other processes will not be able to interfere with locked lines.

Figure 4.1: A single cache line of PLCache

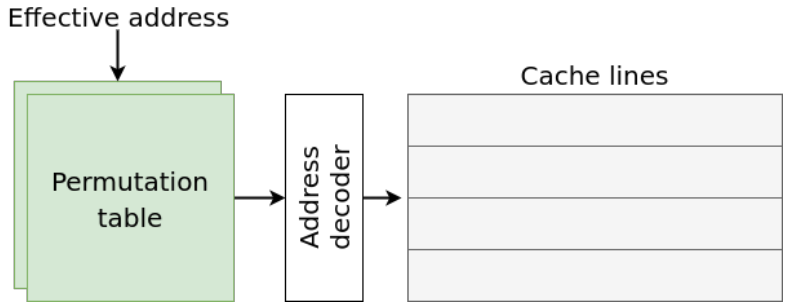

Figure 4.2: Address decoding in RPCache

For implementing the PLCache, the hardware level changes include extra bits for every cache line and a change of the replacement policy. However, PLCache requires addition of special locked-load/store instructions to the ISA. This also means OS has to look over which process gets to use them as a fairness measure. If unchecked, attackers can intentionally lock lines which will hinder performance of other programs. Overuse or abuse of the locking feature can lead to severe performance degradation, if not checked by the OS.

PLCache is better than static partitioning in that it allows locked partitions of the cache to be assigned dynamically. But it has certain drawbacks in terms of implementation.

4.2 Random permutation cache

Random-permutation cache is another cache design proposed by Wang et al in (7). They have added a redirection step in the address decoder of caches which uses a random permutation table as seen in Fig. 4.2. The permutation table essentially randomises the cache line in which an address will be stored. The size of permutation table is larger than cache size (in terms of number of lines) such that there is lesser aliasing in the permutation table. Moreover, the replacement policy is modified to update the permutation table for every replacement, hence an attacker will not be able to decode the mapping.

RPcache is also able to mark sensitive data using "Lock" bits which are derived by page protected bit. This is possible without any modification to ISA hence it is better than PLCache. The only drawback of RPCache is the added step in address decoding, which will increase cache latency by one or two cycles. This may not affect L2 or L3 caches much but it will drastically change performance of L1. To overcome this Wang et al propose optimisations to the gate-level hardware of the decoder. The also propose an improved cache architecture called Newcache (8) which overcomes these issues while not losing in power and performance.

4.3 Intentional Cache pollution

Cache pollution happens when unnecessary data resides in cache and evicts important data which is being used by processes. It happens generally due to poor design and designers will try to avoid it as much as possible, by using smarter replacement policies.

From a security perspective, we can use cache pollution to our advantage by introducing enough noise in a cache side channel such that is hides leaked information. There are multiple ways of intentionally polluting the cache. Two such ways are presented below.

4.3.1 Disruptive Prefetching

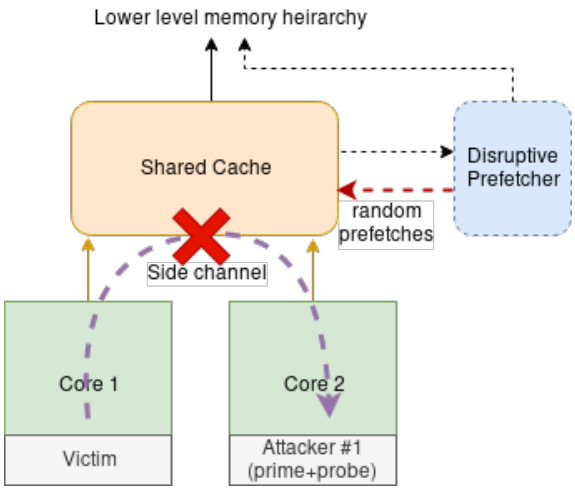

Figure 4.3: Disruptive prefetcher preventing side channel leakage

Pre-fetchers are hardware blocks which were originally designed to hide memory access latency by guessing the need of certain memory address based on previous memory access patterns. Memory locations guessed by the pre-fetcher are loaded into cache so that when the code execution requires that memory, it gives a cache hit instead of miss. Pre-fetchers like Stride pre-fetcher and GHB pre-fetcher are based on finding patterns in the previous memory accesses and guessing that the next locations in that pattern will be accessed. Fuchs et al in (9) introduce additional steps to the prefetchers to increase the randomness in the memor access pattern. They randomise the pattern sequence and degree of prefetching to intentionally pollute the cache with unnecessary data. This will degrade the performance of non-malicious programs by a bit, but will terribly disrupt any side channel established by an attacker. For example, in Prime+Probe attack, the attacker will not know whether the victim or the pre-fetcher evicted its block from the cache, hence it will wrongly trace the memory access of the victim. In the same way, Flush+Reload would get false cache hits which were not caused by the victim.

In Fig. 4.3, there is a side channel established between Victim and Attacker process. The shared cache has a disruptive prefetcher which is continuously introducing random prefetches on every access (prefetcher hit or miss). This is causing the Prime+Probe attack to detect other cache locations which were not accessed by the victim but because of these random prefetches.

4.3.2 Context sensitive decoding

A lot of modern processors use decoders to convert from ISA to an internal instruction representation. Most popularly Intel converts from x86 ISA to microcode using a microcode cache mapping table. Taram et al explore in (10) if a custom decoder can be used to improve the security of certain programs. They use the decoder to introduce decoy instructions in the pipeline. These decoy instructions will change the timing characteristics of the executing program, they will pollute the cache by running decoy loads and will disrupt attackers attempting side channel or timing attacks. Their implementation, as seen in Fig. 4.4 includes adding custom decoder hardware, and a few changes to the microcode mapping table (of which there exists an established update procedure), and a few model specific registers to control the context of the program. They show their method to be effective in stopping I-Cache and D-Cache side channel attacks against RSA and AES.

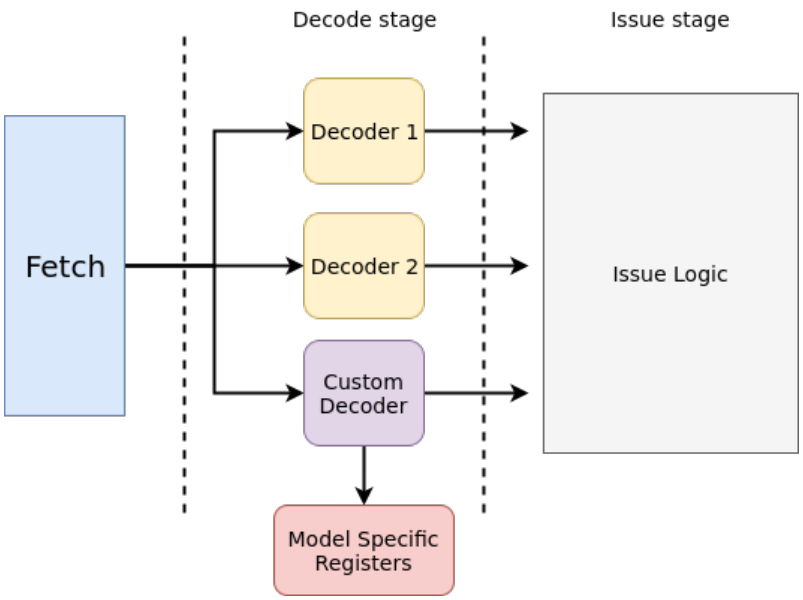

Figure 4.4: Custom decoder for context sensitive decoding

Chapter 5

Attack on Prefetcher

For side-channel attacks to function better and extract data in lesser number of cycles, they need to isolate the victim to execute memory accesses. Any other program which may be running will only interfere with the side-channel. Even if the victim is executing in some other region of code which does not directly implement the security algorithm, memory accesses of that execution will also affect the noise in the side channel. Data prefetchers are also troublesome in this regard because they can generate memory accesses which did not originate from the victim. This ability is utilised by Disruptive Prefetcher as seen in Section 4.3.1.

5.1 Motivation

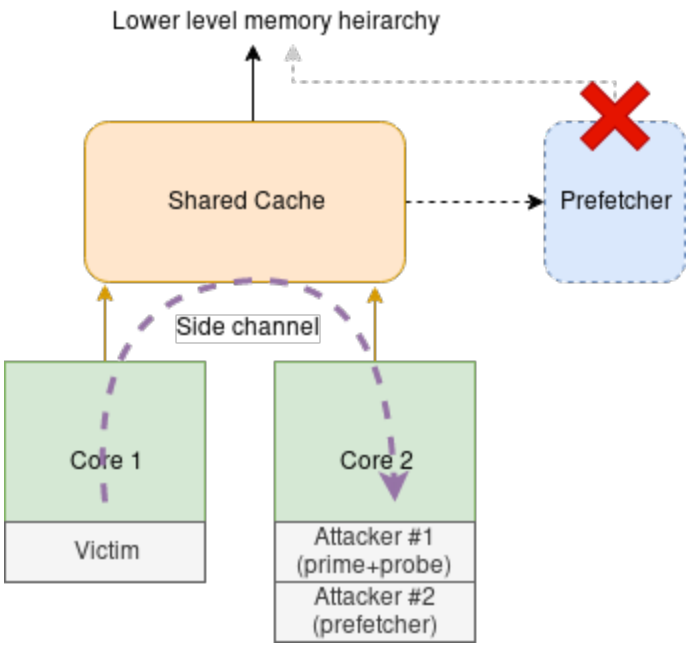

Figure 5.1: Prevent prefetcher from issuing memory accesses

An attacker with reasonable access to the system which the victim is running on has the capability to restrict execution of only the victim, along with isolating the region of code relevant to encryption. But there is no direct way to restrict the prefetcher from generating memory access. A separate attack can be mounted on the prefetcher which prevents it from training on victim addresses and having low confidence to avoid issuing any prefetches. As seen in Fig. 5.2, such an attacker can run simultaneously with a Prime+Probe attacker.

5.2 Implementation

The most basic form of the prefetcher is a Stride Prefetcher. The PC address, last memory address and the last stride and confidence counter is stored in a table and checked whenever there is a cache miss (13). If PC address lookup is successful and the stride matches, the confidence counter is incremented. If the counter is above a certain threshold, we can say that this stride value has gained enough confidence to match the program's true patter. This stride and memory address is used to generate memory accesses separated by the stride value. For simulation, Stride Prefetcher implementation of gem5 (**?**) has been used which is similar in implementation to what is described here (**?**).

The victim program may have multiple load instructions which generate a stride pattern and the PC of these instructions will be recorded by the prefetcher. The attacker tries to prevent each entry of the prefetcher from training on any such load instructions of the victim. If there is a chance that the prefetcher has recorded some load instruction, the attacker tries to evict this entry from the table by placing its own entry. New entries in the prefetcher are initialised with a low confidence value by default. If the victim load instruction is eventually re-entered into the prefetcher table, it will have to retrain to get the confidence above threshold and this will prevent prefetches from being generated for some time. The attacker continuously keeps filling the prefetcher with its own entries so that the victim's entries cannot generate a high enough confidence.

To achieve this, the attacker basically has to bombard the prefetcher with enough load instructions to fill up the whole table. This is done by placing the instructions at many different PC addresses which ensure that atleast one load aliases to every entry in the table. Stride value of access is randomized for every load instruction so that the prefetcher does not generate prefetches for attacker's loads. Memory addresses of the load instructions are also restricted to a single cache line so that the attacker does not interfere with an already running side-channel attack. This is done by keeping the addresses in same 64-byte boundary. To ensure there is a cache miss every time, flushing before load using

`clflush` needs to be done. If the attacker's loads do not generate a cache miss, prefetcher is never accessed. `nop` instructions are used to align loads to the desired PC address.

5.2.1 Full attacker

Listing 5.1 shows a part of the Assembly instructions which are used in the attacker. These instructions show how loads are placed at different PC addresses and how they will be able to fill all entries of the prefetcher. This is called the "full" attacker because it tries to fill up the whole prefetcher table.

Listing 5.1: Assembly showing load misses ar different PCs

```
00000000000006ca <attack>:
 6ce:   8b 58 36        mov    0x36(%rax),%ebx
 6d1:   90              nop
 6d2:   0f ae 78 36     clflush 0x36(%rax)
 6d6:   8b 58 08        mov    0x8(%rax),%ebx
 6d9:   90              nop
 6da:   0f ae 78 08     clflush 0x8(%rax)
 6de:   8b 58 3f        mov    0x3f(%rax),%ebx
 6e1:   90              nop
 6e2:   0f ae 78 3f     clflush 0x3f(%rax)
 6e6:   8b 58 38        mov    0x38(%rax),%ebx
 6e9:   90              nop
 6ea:   0f ae 78 38     clflush 0x38(%rax)
 6ee:   8b 58 20        mov    0x20(%rax),%ebx
 6f1:   90              nop
```

The drawback of this full attack is that it is extremely slow. There are about 256 loads which all generate a cache miss. The long time which the attacker takes to execute leads to gaps in the attack. In these gaps, the victim's load instructions get trained by the prefetcher and generate memory accesses. To remove the newly trained PC addresses, the attack loop has to start again which takes a long time.

5.2.2 Targeted attacker

The lesser the loads which we would need to execute, the better this attack will function. It is possible to conduct offline analysis of victim process and predict which PC addresses would lead to maximum prefetches being generated. If these PC addresses are found,

only they need to be targeted by the attacker. Then the number of loads which need to be executed is reduced from 256 to 32. This can be seen in Listing 5.2.

Listing 5.2: Attacker targeting specific PC addresses

```
00000000000006ca <attack>:
    ...
 6d9:   90                      nop
 6da:   8b 58 0f                mov    0xf(%rax),%ebx
 6dd:   0f ae 78 0f             clflush 0xf(%rax)
 6e1:   90                      nop
 6e2:   90                      nop
 6e3:   90                      nop
 6e4:   8b 58 3c                mov    0x3c(%rax),%ebx
 6e7:   0f ae 78 3c             clflush 0x3c(%rax)
 6eb:   90                      nop
 6ec:   90                      nop
    <nop slide> ...
 6f7:   90                      nop
 6f8:   8b 58 2f                mov    0x2f(%rax),%ebx
 6fb:   0f ae 78 2f             clflush 0x2f(%rax)
 6ff:   90                      nop
    ...
```

Only those load instructions which will alias with the victim's loads are being executed, and the gaps in between are filled with repeating nop instructions called nop slide. This nop slide will introduce a very small delay thus reducing the gap which was present in the full attack.

5.3 Simulation and Results

5.3.1 Simulator setup

Simulator	gem5 X86
Cores	2
L1 Icache	32K 8-way
L1 Dcache	32K 8-way
L2 cache	256K 16-way shared between cores
L2 prefetcher	Stride 64-entry 4-way, confthresh 4

Victim process runs on core1 and attacker runs on core2. Number of prefetches issued are measured for every 1,000,000 instructions of victim process.

5.3.2 Results for benchmarks

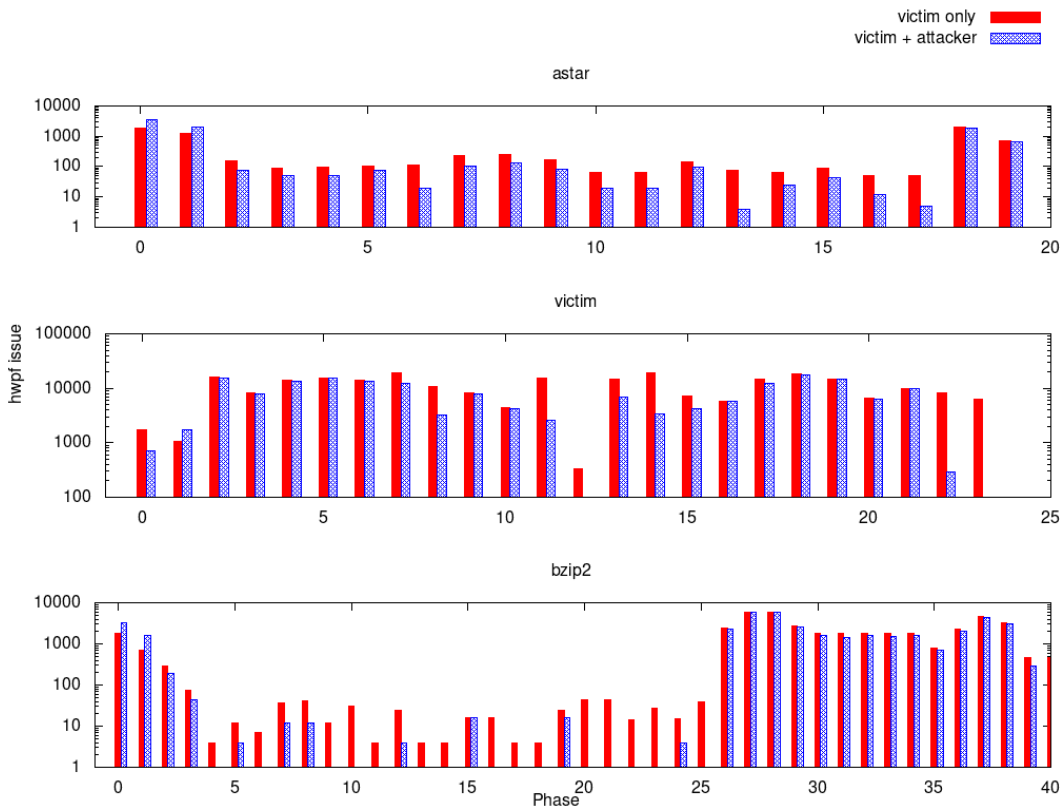

Figure 5.2: Number of prefetches issued on different benchmarks

The full attacker implementation is used for Fig. 5.2 and Fig. 5.3. These results show the drawback of full attacker where it is not able to fully eliminate the prefetches issued. This is because of the long execution time which allows the victim to retrain the prefetcher.

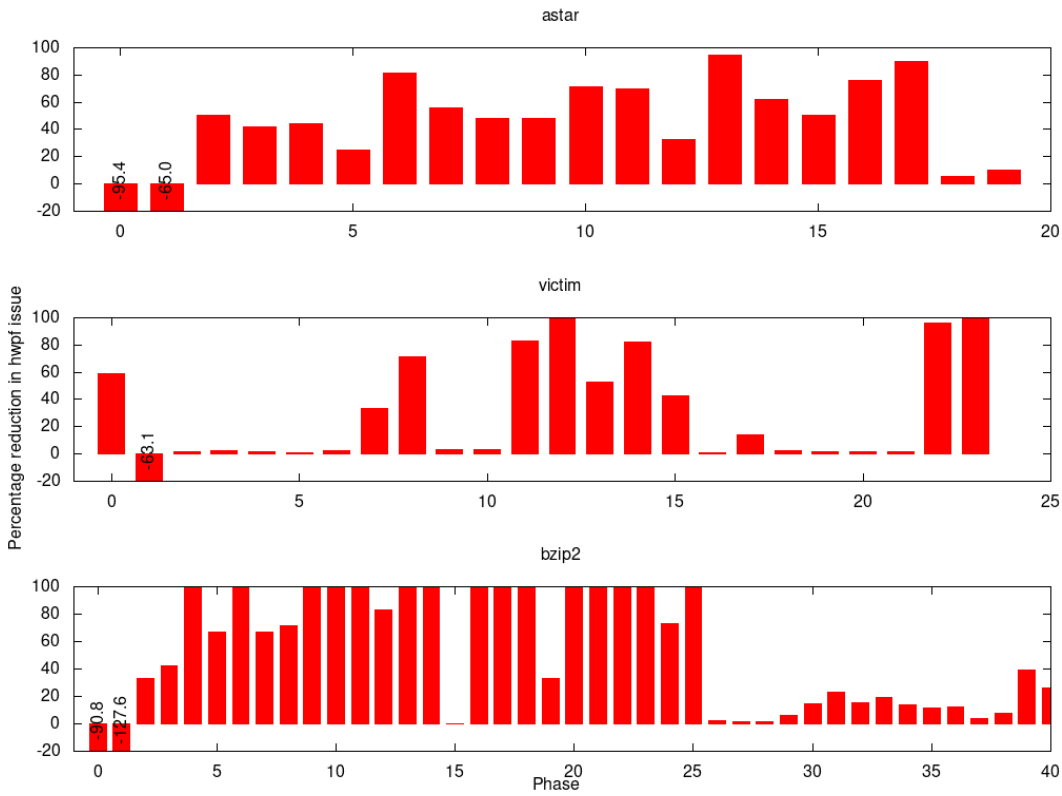

Figure 5.3: Percentage reduction in number of prefetches

5.3.3 Results for AES

An implementation of AES is best to test for prefetches because that is the code that will generate memory accesses if any. The following results in Fig. 5.4 and Fig. 5.5 show number of prefetches and prefetcher table hits vs misses for both the full attacker and targeted attacker. As seen in the Fig. 5.4, a targeted attacker reduces the number of prefetches issued to zero in most of the phases. Fig. 5.5 shows that this is because more prefetcher accesses are leading to PC lookup miss than without attacker.

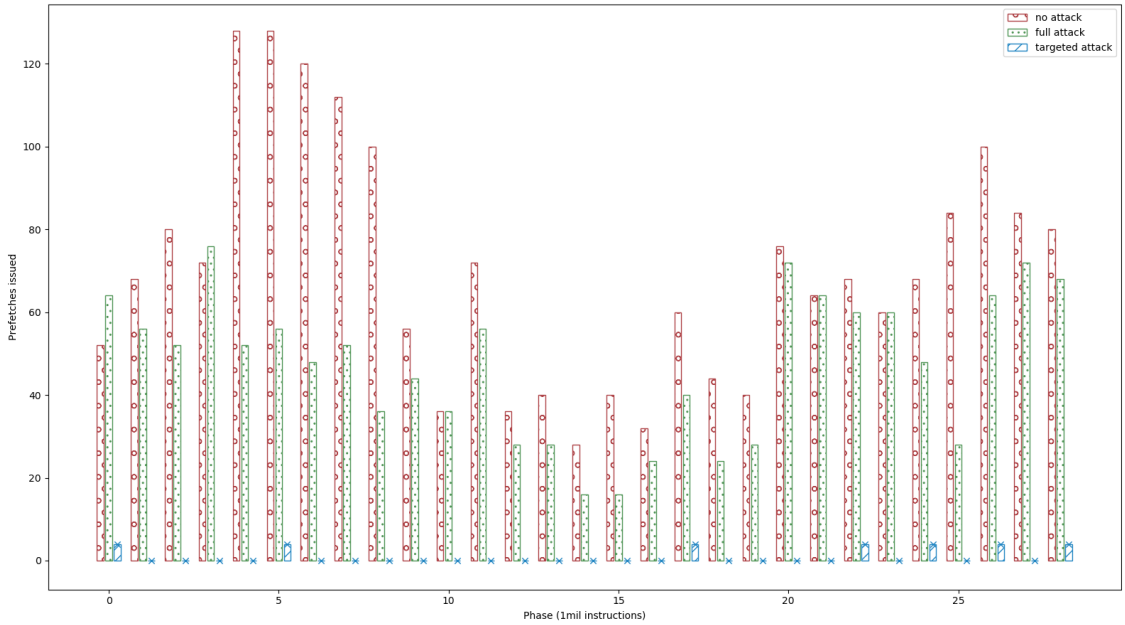

Figure 5.4: Number of prefetches issued on different benchmarks

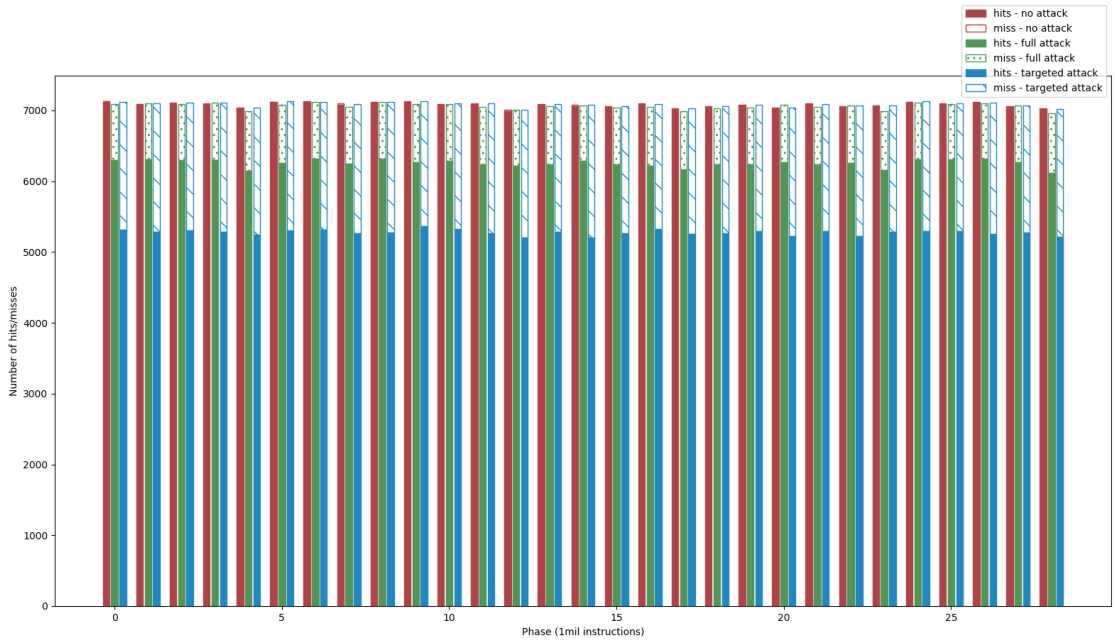

Figure 5.5: Percentage reduction in number of prefetches

Chapter 6

Side-channel using Reorder Buffer

Reorder buffer is used by Out-of-Order cores for in-order retirement. In SMT cores, fairness policy ensures that roughly equal instructions are retired for all threads to maintain equal IPC. Simple implementation is to share retire pipeline width equally among threads. But when one thread is stalled, other thread will use full pipeline width to retire. IPC of other threads will see a slight increase. Data-dependent branch misses and cache misses can be infered from IPC information.

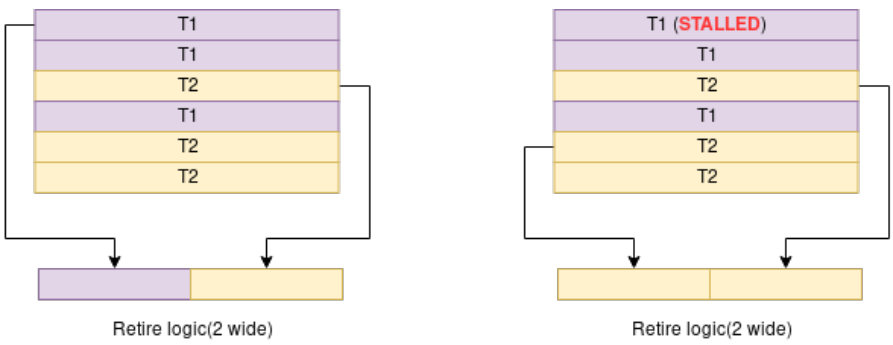

Figure 6.1: Reorder buffer for SMT. When T1 is stalled T2 retires twice as many instructions.

Chapter 7

Conclusion

As we saw above, the recent advances in computer architecture for performance and power gains leads to introduction of a series of leakages from the processor which is a security concern. It is required to design hardware by keeping security in mind as the processors get more and more complex in order to extract higher performance gains. The industry runs a lot of security critical applications which if broken can cause many problems. It is important for us to keep in mind the existance of such side channels to ensure minimal leakage of sensitive data.

References

[1] Chenglu Jin, *Side Channel Attacks*,

[2] D. Page, *Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel*

[3] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, Berk Sunar, *Wait a minute! A fast, Cross-VM attack on AES*

[4] Hoda Naghibijouybari, Khaled N. Khasawneh, Nael Abu-Ghazaleh, *Constructing and Characterizing Covert Channels on GPGPUs*

[5] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos, *Demystifying GPU Microarchitecture through Microbenchmarking*

[6] D. Page, *Partitioned Cache Architecture as a Side-Channel Defence Mechanism*

[7] Zhenghong Wang and Ruby B. Lee, *New Cache Designs for Thwarting Software Cache-based Side Channel Attacks*

[8] Zhenghong Wang and Ruby B. Lee, *A Novel Cache Architecture with Enhanced Performance and Security*

[9] Adi Fuchs, Ruby B. Lee, *Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs*

[10] Mohammadkazem Taram, Ashish Venkat, Dean Tullsen, *Mobilizing the Micro-Ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency*

[11] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, *Meltdown*

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom, *Spectre Attacks: Exploiting Speculative Execution*

[13] J. W. C. Fu, J. H. Patel and B. L. Janssens, *Stride Directed Prefetching In Scalar Processors*, [1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25

[14] The gem5 Simulator. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. May 2011, ACM SIGARCH Computer Architecture News.

[15] http://www.gem5.org/docs/html/stride$_8 cc_s ource.html$