Disabling Prefetcher to Amplify Cache Side Channels

Abstract—**Cache side channels are well known for being effective in extracting data from modern cryptographic ciphers. Attackers try to generate collisions with a victim program sharing the same cache, and study their own cache timings to infer the victim's memory accesses. Some other hardware accessing the cache, e.g. prefetcher, degrades the quality of the side channel by introducing false positives in the attacker's data. This paper describes a method to disable the prefetcher by preventing it from generating memory accesses and interfering with side channels running in the cache. An attacker implementation is designed to work on a Stride Prefetcher. Results show that it is able to significantly reduce the number of prefetches generated to almost 0.**

Index Terms—**security, side channels, prefetcher**

I. Introduction

An attacker program using the cache as a side channel tries to force collisions with the victim program by making accesses which alias to the same cache lines [1]. Time taken for subsequent accesses to these cache lines differs and is used to determine whether there was a successful collision or not. This data is further used to infer whether the victim accessed a particular cache line or not, thus leaking data about the data of the program. Different implementations of the side channel look for either a cache hit or a cache miss as a sign of successful collision. The Prime+Probe attack fills the cache lines in a set with data other than that being accessed by the victim. Any access by the victim to that set will cause attacke's data to be evicted, which will show up during the Probe step as a cache miss [1]. Similarly, the Flush+Reload attack looks for a cache hit to the same data as the victim. A cache hit in the Reload step is infered as successful collision [2].

The attacker assumes a scenario where only the victim is making memory accesses, hence is able to deduce the memory access pattern. If there is another program or hardware making memory accesses, they will surely interfere with the side channel. After obtaining a successful collision, there is no way for the attacker to distinguish whether the source of this collision was truly the victim. Fuchs et al [3] introduce a Disruptive Prefetcher which generates spurious memory accesses, making the victim's accesses indistinguishable for any attacker.

This paper focuses on a way to disable the prefetcher and significantly reduce the number of generated prefetches. With a separate attacker focusing on disabling the prefetcher, it becomes extremely unlikely for the side channel attacker to see a collision with a prefetcher generated access. This enhances the side channel and can enable faster and better data retrieval. The attack implementation has been designed specific to a Stride prefetcher [4]. However, the implementation can be used as-is or easily extended to apply to any PC-indexed prefetcher table.

II. Attack Vectors

A Stride Prefetcher tries to identify load instructions which have a pattern with constant distance between accesses i.e. a fixed stride. The prefetcher table stores entries containing PC address of the load instruction, the last accessed memory address, the stride value and a confidence counter. The table is indexed using the load PC, which leads to aliasing between multiple PCs. Higher the value of the confidence counter, higher is the probability that the next access follows the same stride pattern currently stored. The prefetcher generates memory accesses when it sees an entry with high enough confidence. Every entry needs to be prevented from reaching this condition to disable the prefetcher. This design exposes two attack vectors which the attacker can use to prevent entries from gaining high confidence.

Evict Table Entries: The attacker can keep creating many new entries in the table, and the prefetcher will be forced to evict older entries of the victim which have gained high confidence. When the victim's entry is added again to the table, it will start from a lower default confidence and will have to go through the training phase again. If the victim's entry is quickly evicted by the attacker's entry, it can never gain enough confidence to generate memory accesses.

Decrement Confidence: The prefetcher calculates the new stride value for every access using the last address. When this new stride differs from the last stride stored in the entry, the confidence counter is decremented and the old stride is replaced with the newly calculated one. This helps to keep confidence low for the attacker's entries and ensures that there are not accesses generated due to the attacker.

These two attack vectors are utilised to implement an attacker whose target is to reduce the memory accesses generated by the prefetcher to zero.

III. Attacker Implementation

To create new entries in the table, every load executed by the attacker has to come at a new PC address. A single load inside a loop will only create one new entry. The attacker binary is created such that a large number of load instructions are placed at different PC addresses. There need to be enough load instructions properly located at different PC addresses so that, after aliasing, every location in the prefetcher table is accessed atleast once.

It is generally the case that the prefetcher is accessed only on cache miss. To ensure that the attacker's loads generate a cache miss the memory address is flushed from the cache hierarchy using `clflush` instruction [5].

A. Full Attacker

The full attacker is designed in a way to target the whole prefetcher table, without considering the victim program running. It targets to disable every entry in the table by keeping the confidence value low. Multiple load instructions have to be placed at different PC addresses so that each entry in the table is aliased to atleast once. Considering a set-associative table, the set-indexing bits of the PC are identified. Single load instruction in x86 is of 3 bytes. The corresponding `clflush` instruction is of 4 bytes. An extra `nop` instruction has been added with the pair to round up the PC increment to 8 bytes. A large enough sequence of these set of instructions, with extra `nop` instructions wherever required, is generated to ensure aliasing to every entry in the table. It is important that each entry gets a nearly equal number of hits from the attacker, to be properly effective. The size of 8 bytes of the set of instructions helps in this versus 7 bytes.

```
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
   ...
```

Listing 1. Full Attacker disassembly: load misses at different PCs

Listing 1 shows a part of the disassembly of the binary generated. The full attacker takes time to run a single iteration of the attack because of the repeated cache misses. An attacker targeting a 16-set 4-way prefetcher table requires 128 load instructions. When each of these loads gives a cache miss, the latency of the attacker becomes very high. While one iteration of the attack is running, it is possible that some of the victim's loads can re-enter the table and build up a high enough confidence to generate prefetches. This will be seen in the results in Section V.

B. Targeted Attacker

A faster implementation is required which can quickly evict such notorious loads of the victim program. When the victim program is known, it is possible to predict which loads will be able to re-train the prefetcher very quickly. The victim program is simulated and its memory access pattern is recorded. This pattern when applied to a simulated model of the prefetcher gives an idea about the load instructions which are likely generate the most prefetches. The targeted attacker is tailored to these load instructions and leaves the rest of the prefetcher entries untouched. The targeted attacker is generated by filtering out unnecessary load instructions from the full attacker binary and replacing them by `nop` instructions. This leads to a binary with few load instructions scattered and filled with `nop` slides. A binary generated for hitting 2 load instructions of the victim requires 16 loads compared to the 128 loads of the full attacker. This reduces the latency of the attacker significantly and makes the attacker more effective against a victim program with few notorious loads.

```
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
     ...
```

Listing 2. Targeted attacker disassembly: loads at aliased PCs

Listing 2 shows a part of the disassembly of the targeted attacker binary. The `nop` slides look like they would add some delay in between but that is masked by the cache miss latency caused by the load instruction.

IV. Simulation

Table I shows the configuration of the simulator and included hardware. The victim program runs on core 1 and attacker runs on core 2. The simulator makes measurements for a phase of certain number of instructions. It records the number of prefetches issued, average confidence of the entries, hits and misses to the prefetcher table by victim program.

TABLE I
Simulation setup

Simulator	gem5 X86
Core Type	O3 CPUs 8-wide fetch
Number of 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, confidence threshold 4

V. Results

Figures 2 and 3 show results of testing the full attacker with a benchmark programs *astar*, *bzip2* and a sample program *stride access generator*. It is evident that the full attacker is not very effective in some conditions of the program, which is unacceptable.

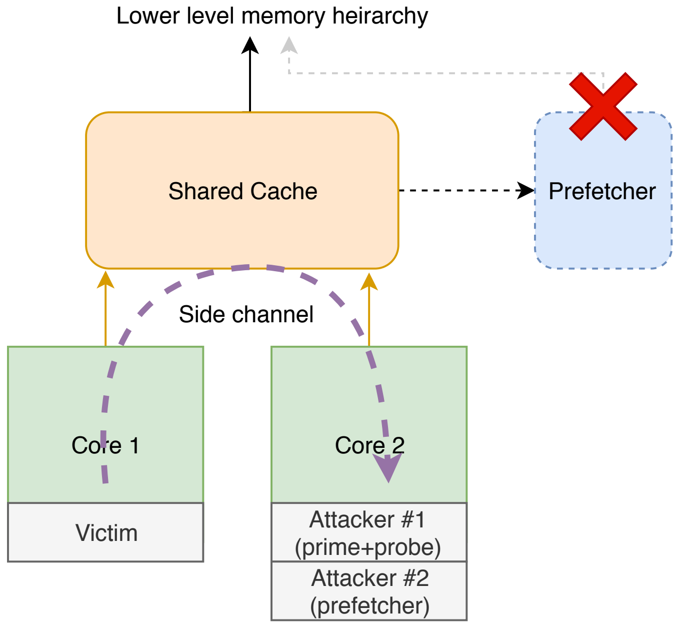

Fig. 1. Setup of attack to disable prefetcher from generating memory accesses

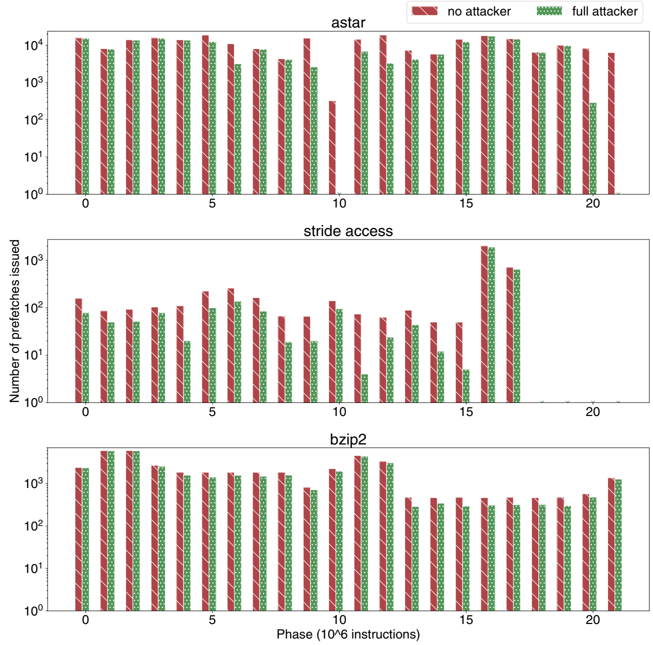

Fig. 2. Number of prefetches issued on different benchmarks

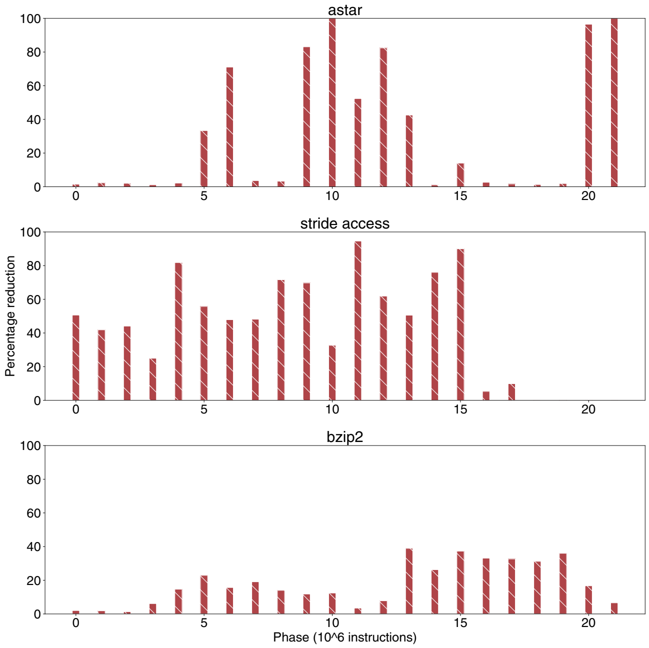

Fig. 3. Percentage reduction in number of prefetches

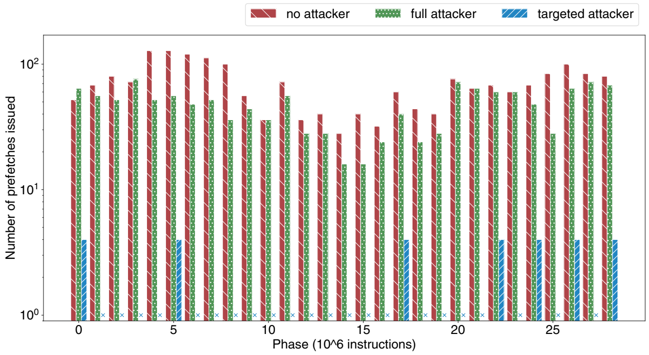

Fig. 4. Comparision of number of prefetches issued by AES program

For more relevant results, further tests are conducted with the OpenSSL implementation of AES algorithm. The AES library function is run repeatedly with random inputs and the same key. The results under various attack scenarios are measured and compared in Figures 4, 5 and 6. Two load instructions are identified for AES victim by using the method outlined in Section III-B. The targeted attacker is tailored to those load PCs. The main observation in Figure 4 is that the targeted attacker is significantly more effective than the full attacker. The full attacker is able to achieve an average reduction of 32%, while the targeted attacker is able to successfully reduce the prefetches to 0. In Figure 5, the average confidence of the 2 sets which targeted prefetcher is attacking is shown. The average confidence with no attacker is 6.9, with full attacker is 5.2 and with targeted attacker is 3.5. The targeted attacker is able to lower confidence below the threshold value of 4 hence is successful in reducing the prefetches.

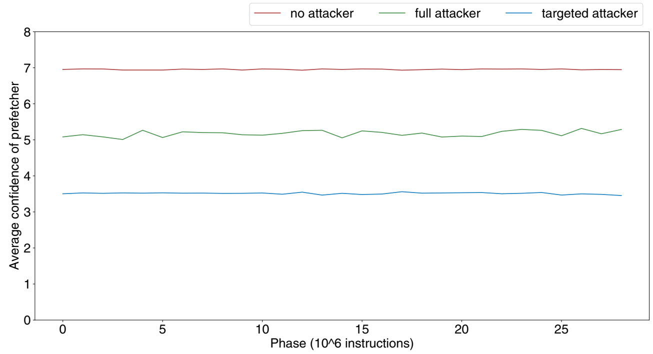

Fig. 5. Comparision of average confidence of prefetcher with AES program

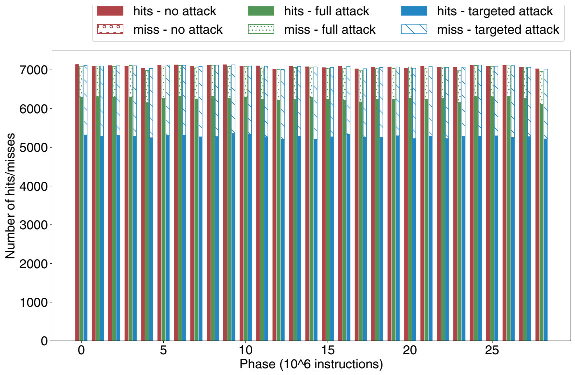

Fig. 6. Comparision of prefetch table hit and miss count by AES program

The effectiveness of targeted attacker can also be seen in Figure 6 as it is able to double the miss rate of the victim program.

DCPT prefetcher: The same full attacker implementation is tested with a DCPT prefetcher. A DCPT Prefetcher [6] is a PC indexed table which uses history buffers to stores deltas of every memory address instead of a single stride value. The history is then used to predict future deltas. A 128 entry fully associative table is added instead of the stride prefetcher in the same setup described in Table I. Figure 7 shows that the full attacker is able to reduce the number of prefetches by 99%. This is because of the random strides generated by the attacker which fill up history buffers of each entry with irrelevant info. The victim's pattern does not get enough time to be recorded completely.

VI. Conclusion

The motivation of this work was to disable the prefetcher from generating memory accesses and prevent it from adding noise to cache side channels. The memory access generated by a prefetcher may or may not be eventually requested by the victim program, hence the prefetcher causes false positives in a cache side channel. When the prefetcher is disabled, the system becomes equivalent to that with no prefetcher present and side channels are amplified. This paper presents two implementations of an attack on the prefetcher. An analysis of the working of a stride prefetcher presents two attack vectors which can be exploited for an attack. Stride prefetchers use the confidence counter of a valid entry in the table to generate prefetches. The attacker is designed such that valid entries of the victim program are regularly evicted from the table, and the confidence of these entries is not allowed to cross the threshold.

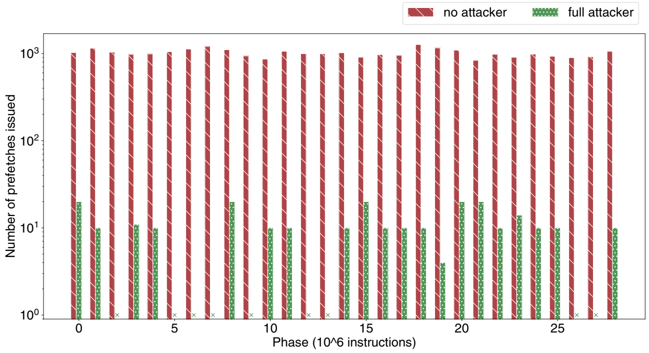

Fig. 7. Comparision of number of prefetches issued by DCPT Prefetcher with AES program

The full attacker is designed to work on the whole prefetcher table. It is only able to reduce the number of prefetches issued by 32% for the AES victim program. The inefficiency of the full attacker is improved in the targeted attacker which only attacks specific entries of the victim program. These entries are identified beforehand by running simulations and the targeted attacker is constructed to target these entries. The targeted attacker is able to reduce the number of prefetches issued to 0. The reason of this significantly better performance can be seen in the plots of average confidence and hit rate.

The full attacker is also tried on a DCPT prefetcher having a fully associative table of 128 entries. The attacker gives a reduction of 99% in number of prefetches issued. However, it is not able to reduce the number to completely 0.

VII. Future Scope

There is scope to build a better attacker which is tailored for history-buffer based prefetchers like the DCPT prefetcher. Further tests can be run by testing the attacker in parallel with a side channel prime+probe attacker to see the impact. The effectiveness of this attacker on ciphers apart from AES also can be explored. A similar analysis of security-oriented prefetcher designs, e.g. the Disruptive prefetcher, can be conducted. It may expose certain weaknesses of the design.

References

[1] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of aes,” in *Cryptographers’ track at the RSA conference*, pp. 1–20, Springer, 2006.

[2] C. Percival, “Cache missing for fun and profit,” 2005.

[3] A. Fuchs and R. B. Lee, “Disruptive prefetching: impact on side-channel attacks and cache designs,” in *Proceedings of the 8th ACM International Systems and Storage Conference*, p. 14, ACM, 2015.

[4] J. W. C. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, (Los Alamitos, CA, USA), pp. 102–110, IEEE Computer Society Press, 1992.

[5] Intel®, “Intel® 64 and ia-32 architecturessoftware developer’s manual.”

[6] M. Grannaes, M. Jahre, and L. Natvig, “Storage efficient hardware prefetching using delta-correlating prediction tables,” *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–16, 2011.