IEICE TRANS. INF. & SYST., VOL.E104–D, NO.8 AUGUST 2021

LETTER

DCUIP Poisoning Attack in Intel x86 Processors

Youngjoo SHIN[†a)], ***Member***

SUMMARY Cache prefetching technique brings huge benefits to performance improvement, but it comes at the cost of microarchitectural security in processors. In this letter, we deep dive into internal workings of a DCUIP prefetcher, which is one of prefetchers equipped in Intel processors. We discover that a DCUIP table is shared among different execution contexts in hyperthreading-enabled processors, which leads to another microarchitectural vulnerability. By exploiting the vulnerability, we propose a DCUIP poisoning attack. We demonstrate an AES encryption key can be extracted from an AES-NI implementation by mounting the proposed attack.

key words: *hardware prefetching, Intel DCUIP prefetcher, microarchitectural side-channel attack, poisoning attack*

1. Introduction

Cache prefetching is a processor optimization technique that boosts up software performance by speculatively fetching data from memory in advance of the software execution. Once prefetched, the data resides on a cache and further accesses to the data will be directly served from the cache with low latency. As cache prefetching brings huge benefits to the overall performance improvement, processor vendors are now aggressively employing prefecthers in their CPU. For instance, Intel x86 processors are equipped with four prefetching units in every core trying to catch all the prefetchable memory access patterns [1].

However, such performance benefits from the prefetching technique comes at the cost of security in software systems. As prefetching activities reflect on memory access patterns of a victim's program, an attacker can infer the access pattern by observing whether certain cache lines have been prefetched via cache side-channel analysis techniques. For instance, recent work by Shin *et al.* demonstrated how to extract an ECDH private key by observing prefetching activities on a victim's program [2].

In this letter, we go further than the previous work: we dissect internal workings of Intel prefetchers, and based on the analysis result we present a new microarchitectural attack by *poisoning the prefetching behavior*. In particular, we focus on an DCUIP prefetcher, which is one of Intel prefetchers located on L1D cache. DCUIP prefetcher looks for strided memory access patterns and tries to prefetch a next cache line predicted to be accessed based on the observation. For this purpose, the DCUIP prefetcher maintains a history table, which is referred to as *DCUIP table* in the rest of this letter, to trace multiple access patterns in a program. Each entry of the DCUIP table contains a state information for an observed memory access: the state includes a tag of the pattern, the last accessed address and the observed stride with its confidence.

We discover from our analysis that the DCUIP table is shared among logical threads on a CPU core. That is, a DCUIP state of one program may have influence on the other in hyperthreading-enabled processors, provided that both programs are running on the same physical core. By exploiting the shared table, an attacker is able to poison a victim's DCUIP state to make the victim to prefetch a secret data from memory, even the data which is not supposed to be accessed again after loaded to a register.

Based on our discovery, we elaborate a DCUIP poisoning attack against a victim application that performs AES encryption using AES-NI extension. The victim executes encryption with an AES key, which is located on a register. Hence, once a key has been loaded to the register, the victim never accesses the key from memory during the entire execution. The DCUIP poisoning attack allows an attacker to obtain the victim's secret key from memory through microarchitectural fill buffer data sampling (MFBDS) analysis.

The remainder of this letter is organized as follows. In Sect. 2, we provide some background knowledge about cache prefetching. In Sect. 3, we explain our analysis on an DCUIP prefetching algorithm in Intel x86 processor. In Sect. 4, we describe the proposed attack in detail as well as our evaluation results on the attack. In Sect. 5, we discuss a feasibility of the attack and present some countermeasures. Finally, in Sect. 6, we conclude this letter by summarizing our work.

2. Cache Prefetching

Cache prefetching is an optimization technique to reduce delays caused by cache miss. By predicting memory access patterns, it tries to fetch data from a memory to a cache before it is needed. As there are a diversity of applications with various access patterns, modern processors employ a number of hardware-based cache prefetching techniques to cover the wide range of applications (See Table 1). DCU prefetching performs prefetching the next line N+1 if an access to line N has been detected. Adjacent-line prefetching

Manuscript received November 27, 2020.
Manuscript revised February 25, 2021.
Manuscript publicized May 13, 2021.
[†]The author is with School of Cybersecurity, Korea University, Seoul, Korea.
a) E-mail: syoungjoo@korea.ac.kr
DOI: 10.1587/transinf.2020EDL8148

Table 1 List of cache prefetching units in Intel processors

Name	Algorithm	Cache level
Streamer	Stream	L2
Spatial prefetcher	Adjacent-line	L2
DCU prefetcher	Next-line	L1
DCUIP prefetcher	Stride	L1

utilizes spatial locality when fetching another line adjacent to the accessed cache line N. Stream prefetching prefetches a number of memory lines ahead by assuming that memory accesses are a part of data streaming. DCUIP prefetching [3] traces each load instruction in a program to figure out whether a sequence of memory accesses forms a pattern $a, a+S, a+2S, \ldots$ with a base address a and a regular stride S. Once detected, it performs prefetching the next memory addresses based on the observed stride. The DCUIP prefetcher maintains a DCUIP table, which comprises of table entries indexed with PC tags. Each entry includes a prefetching information such as the last address a, stride S, and a confidence value c. In this letter, we are mainly interested in the DCUIP prefetcher.

3. Diving into DCUIP Prefetcher

In this section, we analyze the internal workings of DCUIP prefetcher by conducting several experiments. Specifically, our aim is to investigate whether any internal states of the prefetcher are shared among different execution contexts and if so, how does the shared state affect the cache behavior of other context.

3.1 Experiment Design

Experimental setup. The experiments were conducted on an Intel i9-10900 (Comet Lake) processor running Ubuntu 20.04.01 LTS 64-bit Linux. The experimental setup consists of two programs acting as a spy and a victim. We construct the spy process as a fork of the victim so that they share the code base in their own address space. In particular, the spy and victim commonly have a 4KB-sized array *g_mem* of 64 elements, which is aligned with a memory page, as well as a function *mem_access*() that performs lookup to the array. The spy initializes *g_mem* with arbitrary values immediately after being forked. This renders a page of the array duplicated while keeping its virtual address aligned with that of the victim.

In our experiment, the spy and victim programs run concurrently on the same host while logical cores on which they execute are controlled by CPU pinning. In order to exclude any possible side effects due to other prefetching activities, we disable all the prefechers except DCUIP via `msrtools` [4].

Algorithms. The spy and victim execute memory operations on *g_mem* according to Algorithm 1 and 2, respectively. The spy program takes two inputs S_s and E_s, both of which are indices of the array *g_mem*, where $E_s > S_s$. It first evicts all the corresponding cache lines ranging from S_s to $E_s - 1$ by using a `clflush` instruction. Then, the spy makes a series of memory accesses to *g_mem* with given indices from S_s to $E_s - 1$. As the memory is accessed with regular stride pattern, it will influence on the status of DCUIP and induce cache prefetching.

Algorithm 1 Spy memory access

Input: S_s, E_s ▷ S_s and E_s ($E_s > S_s$) are indices of *g_mem*
Output: None
1: **procedure** SPY(S_s, E_s)
2: $\forall_{line \in \{S_s, \ldots, E_s - 1\}}$ flush(line)
3: **for** line ← S_s **to** E_s-1 **do**
4: mem_access (line)
5: **end for**
6: **end procedure**

Algorithm 2 Victim memory access

Input: S_v, E_v ▷ S_v and E_v ($E_v > S_v$) are indices of *g_mem*
Output: Result
1: **procedure** VICTIM(S_v, E_v)
2: $\forall_{line \in \{S_v, \ldots, E_v\}}$ flush(line)
3: **for** line ← S_v **to** E_v-1 **do**
4: mem_access (line)
5: **end for**
6: t ← mem_access(E_v)
7: **if** t < threshold **then**
8: Result ← prefetched
9: **else**
10: Result ← not_prefetched
11: **end if**
12: **return** Result
13: **end procedure**

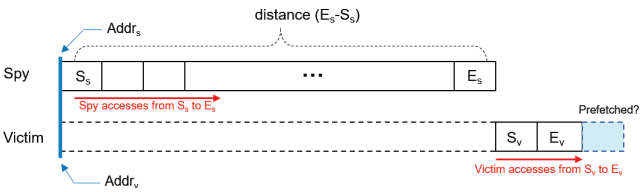

Fig. 1 Spy and victim algorithms in the experiment ($Addr_s$=$Addr_v$)

Like the spy, the victim's algorithm also takes two indices S_v and E_v ($E_v > S_v$) of the array *g_mem* as inputs. After initializing all the corresponding cache lines by eviction, it makes regular accesses to *g_mem* with indices from S_v to $E_v - 1$. Then, the victim checks whether the E_v-th cache line, located adjacent to the line of $E_v - 1$, has been prefetched after the memory operations. The target line (*i.e.,* E_v) is decided to be prefetched if its access latency (t) is below the threshold. The victim's algorithm returns the prefetching result as an output.

3.2 Experimental Result

We measured the hit rate and access latency of the E_v-th cache line while concurrently running the spy and victim programs (of Algorithms 1 and 2, respectively). In the experiments, input arguments of these algorithms were configured as shown in Fig. 1. In particular, S_s for the spy program

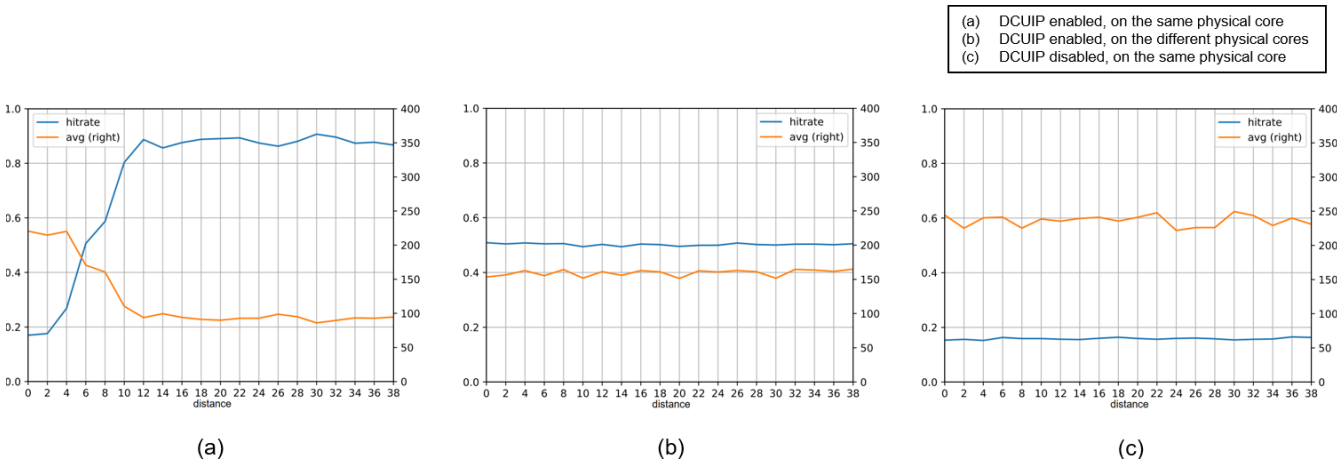

Fig. 2 Experimental result

is set to a base index of the array *g_mem* in spy's address space (referred to as $\texttt{Addr}_s$). For the victim, we configured S_v to be the index right next to E_s (*i.e.*, $S_v = E_s + 1$) and $E_v = S_v + 1$. Note that Addr_v, the base address of *g_mem* in the victim's virtual address space, is the same as Addr_s whereas they differ in their physical addresses.

We conducted the experiment by varying E_s for the purpose of identifying any correlations between the distance (*i.e.*, $E_s - S_s$) of the memory accesses on the spy-side and the prefetching rate on the victim-side. Figure 2 shows the experimental result. The term `'hitrate'` in the graph refers to the prefetching rate of the target (*i.e.*, E_v+1-th line) as a result of the execution. The term `'avg'` refers to the averaged latency of access to the target line. As shown in Fig. 2 (a), the hit rate (as well as the corresponding averaged access latency) is highly correlated with the distance in a range of the distance from 0 to 12. The hitrate consistently remains high (*i.e.*, > 0.8) with distances longer than 12, which indicates that the target line is highly likely to be on cache as a result of prefetching. We confirm from the results shown in Fig. 2 (b) and Fig. 2 (c) that such correlation is only observable when both the spy and victim run on the same physical core and DCUIP prefetching is enabled on that core.

4. DCUIP Poisoning Attack

In this section, we present an DCUIP poisoning attack, a new microarchitectural attack that exploits a shared state of DCUIP to leak secret from a victim.

4.1 Threat Model

As in other microarchitectural attacks proposed in the literature, we assume that both a victim and an attacker are co-located sharing a physical core in the same host. The victim is required to contain a loop in the code that repeatedly performs memory access to a lookup table (LUT); otherwise, DCUIP will be barely active during the attack. Note that this requisite in our attack is reasonable as constructing a loop structure with an LUT is a common practice of implementing practical applications.

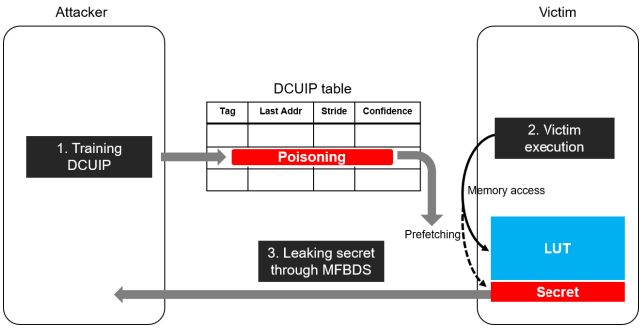

Fig. 3 DCUIP poisoning attack overview

We also suppose that a secret value, in which the attacker is interested, resides at the location adjacent to the LUT in the victim's memory. The secret is hardly accessed by load instructions during the victim's execution. For instance, we consider an AES key used by AES-NI implementations: once the key has been loaded from memory to an AES-NI register, the key will be never accessed again afterward.

The attacker aims to learn the secret from the victim. As the secret is seldom loaded from memory in a normal execution path, the attacker tries to poison the DCUIP so that the secret is being prefetched to cache. Regarding the attacker's ability, we suppose that no information about the victim is given to the attacker except the information about the victim's program binary. In addition, we suppose that no physical memory is shared by the attacker and the victim.

4.2 The Proposed Attack

The overall process of the DCUIP poisoning attack is illustrated in Fig. 3. The attack proceeds in three phases as follows.

Phase 1: Training DCUIP. The attacker first attempts to poison a DCUIP table which is shared with the victim. For

this purpose, he/she trains the DCUIP by using Algorithm 1 described in the previous section with his/her own lookup table. As a result of training, a confidence value for the corresponding entry in the DCUIP table will increase enough to trigger prefetching in the context of the victim.

Phase 2: Victim execution. The attacker controls the victim to begin executing its program, in which internally performs memory accesses to the LUT. If the attacker is incapable of control (*i.e.,* in case of the victim not synchronized with the attacker), he/she waits for the victim to complete the execution. When the victim performs memory accesses to LUT, it is highly likely to trigger prefetching memory address next to the table, where the secret is stored, as its confidence level has been raised by the attacker.

Phase 3: Leaking secret. The secret data is going to be prefetched from memory to L1 cache as a result of the previous phases. In order to complete loading to the cache, a memory subsystem has to temporally keep the data in a line fill buffer (LFB). Hence, the attacker is able to leak the secret from the LFB through a microarchitectural fill buffer data sampling (MFBDS) technique [5], [6] while the data resides in the buffer.

4.3 Case Study: Leaking an AES-NI Key

We now describe the effectiveness of DCUIP poisoning attack by demonstrating a practical attack that leaks an AES key from AES-NI implementation.

Victim implementation. To demonstrate the attack, we built a victim application that encrypts multiple blocks of plaintexts by using an AES-128 algorithm with an ECB mode. The AES encryption was implemented using AES-NI, an extension to the x86 instruction set architecture that accelerates AES algorithms, as follows.

```
aes128_encrypt:
    # Round keys are stored on %xmm5-%xmm15
    # Plaintext block is stored on %xmm0
    pxor       %xmm5,  %xmm0
    aesenc     %xmm6,  %xmm0
    aesenc     %xmm7,  %xmm0
    aesenc     %xmm8,  %xmm0
    aesenc     %xmm9,  %xmm0
    aesenc     %xmm10, %xmm0
    aesenc     %xmm11, %xmm0
    aesenc     %xmm12, %xmm0
    aesenc     %xmm13, %xmm0
    aesenc     %xmm14, %xmm0
    aesenclast %xmm15, %xmm0
```

AES-NI allows the application to process one round of AES encryption with a single x86 instruction. That is, each round is executed with two 16-byte operand registers (*e.g.,* `xmm0` - `xmm15`), one for the round key and the other for the internal state of AES. The victim application first precomputes round keys $R_0, R_1, \ldots, R_{10}$ from a AES-128 secret key, where R_0 comes from the AES-128 key itself. After the key expansion has been completed, all the round keys are stored on a memory buffer. At the beginning of the execution, the precomputed round keys are loaded from the buffer to `xmm` registers; once loaded to the registers, the round keys in the memory are never accessed during the entire execution. The victim repeatedly performs encryption for each plaintext block in a `for`-loop statement. For our attack, we make sure that the round key R_0 in the memory is located adjacent to the plaintext buffer.

Methodology. Our spy program targets the initial round key (*i.e.,* R_0) of the victim among all the round keys. In the experiment, a spy and victim programs are running on two cores, say C_1 and C_2, which are logically separated by hyperthreading but are physically on the same core. We utilized CPU pining, as mentioned in Sect. 3, to statically assign the cores to each program. The procedure for the experiment consists of several steps, which are presented as follows:

- (S_1) In the first step, we have the victim program run on C_1. The victim keeps running by repeatedly executing the function `aes128_encrypt()` during the experiment.
- (S_2) Next, we have the spy program run on C_2. In this step, the spy executes N ($N > 0$) times of DCUIP poisoning attacks against the victim program. For each execution, it records the result of the attack (*i.e.,* an extracted value from the victim) to a log file.
- (S_3) Finally, we stop running the victim program and then exit the procedure.

Attack result. We performed the above attack procedure on an Intel Xeon E3-1275v6 (Kaby Lake) processor running Ubuntu 18.04 LTS 64-bit Linux. In the step S_2, we set N to $N = 100{,}000$. Consequently, we obtained a total of 100,000 extracted values at the end of the procedure. We found that 95,427 values of the obtained results match with the correct value of the secret (*i.e.,* R_0) of the victim application. This allows us to conclude that the DCUIP poisoning attack successfully extracts the AES key from the victim with a success rate of at least 95%.

5. Discussion

In this section, we discuss the feasibility of the DCUIP poisoning attack. In addition, we present several countermeasures to mitigate the proposed attack.

5.1 Feasibility of the Proposed Attack

The feasibility of the DCUIP poisoning attack actually depends on how a victim application is implemented, which is mainly affected by software development practices. The victim AES-NI application implemented in the previous section is intentionally built so that the secret (*i.e.,* the AES key) is located adjacent to the plaintext buffer. Although it is not always the case that applications are built with such a memory layout in practice, we believe that not a few of practical applications are vulnerable to our attack. Indeed, it is not hard to find some AES-NI open sources [7], [8] that have

the same memory layout as our victim application.

Note that vulnerable applications are not confined to AES-NI implementations. Any cryptographic applications that use LUTs in their implementations (*e.g.,* T-table-based AES or squaring operations in elliptic curve cryptography) will be vulnerable to the DCUIP poisoning attack if a secret data (*e.g.,* a private key or any secret-sensitive values) resides adjacently to the LUT in the memory. Software developers who have little concern about this security threats are highly likely to implement applications vulnerable to this attack.

5.2 Countermeasures

We can mitigate the proposed attack by using the following countermeasures.

Secure software development. The root cause of the vulnerability is that a victim application has a memory layout where a secret data is located adjacent to an LUT. Hence, in order to protect software from the DCUIP poisoning attack, software developers should avoid an implementation that has adjacent placement of a secret and an LUT in the memory. The simplest way is to just put a dummy variable between these two objects.

System-level mitigations. As the DCUIP poisoning attack exploits a prefetching activity in recent Intel processors, we can defend vulnerable software from the attack by disabling the prefetcher. Intel processors provide a method to turn off hardware prefetchers by manipulating a model specific register via software such as `msr-tools` [4]. Another solution is to disable hyperthreading of the Intel processors so as to prevent the DCUIP table from being shared between an attacker and a victim.

6. Conclusion

In this letter, we dissected the internal workings of a DCUIP prefetcher, which is one of cache prefetching units equipped in Intel processors. As a result, we discovered that a DCUIP table is shared among different contexts in hyperthreading-enabled processors. We introduced a DCUIP poisoning attack, a new microarchitectural attack in Intel processors. We demonstrated how to extract an AES encryption key from a victim application that performs AES encryption with AES-NI extension by mounting our attack.

Acknowledgements

This work was supported by an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation).

References

[1] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," Tech. Rep., 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

[2] Y. Shin, H.C. Kim, D. Kwon, J.H. Jeong, and J. Hur, "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage," Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.131–145, 2018.

[3] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," IEEE Transactions on Computers, vol.44, no.5, pp.609–623, 1995.

[4] V. Viswanathan, "Disclosure of H/W prefetcher control on some Intel processors," 2014. [Online]. Available: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," Proc. 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.753–768, 2019.

[6] S.V. Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-Flight Data Load," Proc. 2019 IEEE Symposium on Security and Privacy, pp.88–105, 2019.

[7] A. Kasbekar, "AES-NI Encryption," 2021. [Online]. Available: https://github.com/kasbekarameya/AES-NI-Encryption/blob/master/AES-NI Encryption.c

[8] Acapola, "AES128 how-to using GCC and Intel AES-NI," 2021. [Online]. Available: https://gist.github.com/acapola/d5b940da024080dfaf5f