PASS-P: Performance and Security Sensitive Dynamic Cache Partitioning

Nirmal Kumar Boran[a], Pranil Joshi[b] and Virendra Singh
Computer Architecture and Dependable Systems Lab (CADSL),
Indian Institute of Technology Bombay, India

Keywords: Hardware Security, Cache-based Side-channel Attacks, Cache Partitioning.

Abstract: Cache-based side-channel attacks can cause security breaches like extraction of private keys from various encryption algorithms. Static cache partitioning protocols are widely known to prevent such side-channel attacks. However, because static partitioning protocols exhibit poor program performance, dynamic partitioning techniques are preferably used in modern systems. This work exposes the vulnerability of dynamic partitioning protocols such as UCP (Utility-based Cache Partitioning) and SecDCP (Secure Dynamic Cache Partitioning) to well-known side-channel attacks. We then propose PASS-P protocol which prevents such side-channel attacks without compromising on performance. PASS-P, when implemented to secure the widely used UCP protocol, results in an average performance drop of only 0.35%. Compared to the inherently secure static partitioning protocol, PASS-P improves performance by up to 29% (33.4%) and on an average 7.2% (10.6%) in pairs of memory-intensive benchmarks when implemented on the shared L3 (L2) cache.

1 INTRODUCTION

With increasing utilization of computing systems in all domains, security of the systems which handle people's private data and communication has become vital. Extensive research is being carried out to identify security flaws across the stack from hardware to software, and then to come up with novel ideas to fix them.While several ideas have been proposed to to mitigate the exposed vulnerabilities, we limit ourselves to discussion of hardware security in the present work.

Prior work in the field of hardware security has shown the security vulnerabilities in such systems, particularly in the form of side-channel attacks (Percival, 2005; Wang and Lee, 2007; Kong et al., 2008; Ashokkumar et al., 2016; Tromer et al., 2010; Boran et al., 2021). In such attacks, the attacker program exploits specific unintended effects of the victim program. For instance, by using the technique in (Bernstein, 2005), it was shown that an attacker can deduce the private encryption key of the AES (Advanced Encryption Standard) protocol. It did so by exploiting the fact that the total execution time of the algorithm is input-dependent. The different modes in which an attack can be mounted are referred to as 'channels'. These channels include analysis of execution time, memory accesses, power consumption and electromagnetic radiation of the hardware resources being used by the victim program. PASS-P, in particular deals with the kind of attacks in which the attacker tries to analyze the memory accesses made by the victim to find out which parts of the victim program have been executed. Flush+Reload (Yarom and Falkner, 2014) and Prime+Probe (Liu et al., 2015) are side-channel attacks that use differential cache access timing-analysis on lines modified by the victim process. The Flush+Reload technique flushes specific lines from each cache set and then tries to reload the flushed addresses, while the Prime+Probe technique fills the cache sets with the attacker's data and then tries to access the filled data. Because of the difference in memory access latency in cases of cache hit and cache miss, the attacker can deduce the addresses accessed by the victim.

Some recent work (Yao et al., 2019; Yan et al., 2016) has shown novel ways for such attacks to be detected by the system at run-time. COTSknight (Yao et al., 2019) tries to capture the cache occupancy patterns of running processes to identify suspicious applications that could pose a security risk. ReplayCon-

[a] https://orcid.org/0000-0003-3942-7899
[b] https://orcid.org/0000-0001-7013-3034

Boran, N., Joshi, P. and Singh, V.
PASS-P: Performance and Security Sensitive Dynamic Cache Partitioning.
DOI: 10.5220/0011336900003283
In *Proceedings of the 19th International Conference on Security and Cryptography (SECRYPT 2022)*, pages 443-450
ISBN: 978-989-758-590-6; ISSN: 2184-7711

fusion (Yan et al., 2016) replays a program's execution with a different cache address mapping to discern cache miss patterns.

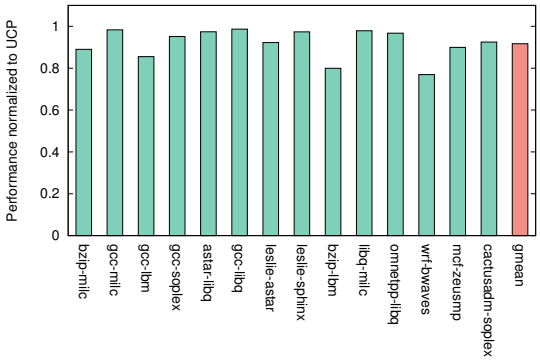

Figure 1: Speedup of static partitioning scheme normalized to UCP for memory-intensive benchmark pairs (for Configuration 1 of Table 1).

On the other hand, there has also been extensive research on mitigating such cache-based side-channel attacks. Approaches to cache security can be broadly classified into two types: cache partitioning and cache randomization (Wang and Lee, 2007). PASS-P follows the first kind of approach. One of the naïve partitioning techniques proposed to do this is static partitioning (Page, 2005). Within each cache set, it enforces a fixed partitioning of the lines amongst the simultaneously running processes. Since no cache resources are shared by processes in this technique, it guarantees security against cache-based side-channel attacks. However, static partitioning comes with a heavy performance penalty because many lines in the cache set remain under-utilized (Wang et al., 2016). Cache access behavior of a program can change during run-time, and static partitioning fails to adapt to this change. To improve performance in a multi-processor system, several dynamic cache partitioning (DCP) methods (Wang et al., 2016; Qureshi and Patt, 2006; Domnitser et al., 2012a; Sanchez and Kozyrakis, 2012; Xie and Loh, 2009) have been proposed. Utility-based Cache Partitioning (UCP) (Qureshi and Patt, 2006) dynamically partitions the LLC in order to maximize the total utility of cache lines for all the running processes. Our evaluation (Figure 1) shows that the static partitioning suffers an average performance degradation of 8.3% with respect to UCP for memory intensive benchmark pairs. Figure 1 also shows that UCP performs better than static partitioning in all experiments.

While these DCP protocols have significant performance advantages, we will show in Section 3 that they are susceptible to cache-based side-channel attacks. For mounting any side-channel attack, the following conditions (Wang and Lee, 2007) must be satisfied: 1) attacker and victim processes must share a resource; 2) both should be able to change the state of the shared resource; and 3) the attacker should be able to detect the changes made by the victim in the shared resource. The LLC is typically shared amongst all running processes. Since LLC lines are reallocated periodically by UCP, all of these conditions are satisfied and an attack can be mounted through the shared LLC. Thus, our goal is to provide security to DCP schemes without incurring a significant performance penalty.

The current work proposes the following contributions:

- We describe a security vulnerability in DCP protocols and SecDCP (Wang et al., 2016) via the shared last level cache. The vulnerability allows an attacker to determine memory accesses made by 'victim' process while running simultaneously on the same core or on another core in multi-core system.
- We propose PASS-P, a protocol that mitigates this vulnerability by invalidating cache lines.
- To recover the performance loss due to invalidation, PASS-P uses the novel Modified-LRU reallocation policy. Our detailed evaluation shows that PASS-P regains the performance lost and performs comparably to UCP.

While this work focuses on how PASS-P can be used to make UCP secure, we believe that our techniques can benefit most DCP schemes which are susceptible to the threat model described in Section 3.1.

The rest of the paper is organized as follows: Section 2 summarizes the relevant prior work in this field. Section 3 gives a brief outline of the UCP protocol, and describes how it is susceptible to side-channel attacks. Section 4 discusses the proposed PASS-P method. Section 5 compares the performance of PASS-P with static partitioning and vanilla UCP. Section 6 concludes our work.

2 PRIOR WORK

Prior work for mitigation of cache-based side-channel attacks can be broadly classified into two approaches: (1) cache randomization and (2) cache partitioning (Wang and Lee, 2007). In the first approach (Wang and Lee, 2007; Qureshi, 2018), the address mapping from main memory to the cache subsystem is randomized so that no process can precisely detect the accesses made by any other process. RPCache (Wang and Lee, 2007) achieves this using a permutation table

to achieve this randomization. In CEASER (Qureshi, 2018), a low-latency block cipher is used to encrypt the address used to access cache. In the second approach, the side channel is sealed by partitioning the cache system amongst the running processes, such that no process can access the cache lines of any other process. PASS-P is based on this second kind of approach.

As discussed in Section 1, the method of static partitioning (Page, 2005) ensures security against all cache-based side-channel attacks. However, it comes with a significant performance degradation because of its inefficient use of the cache system. On the other hand, dynamic cache partitioning (DCP) protocols, which were developed to improve performance, are insecure. Prior research has recognized this insecurity and tried to mitigate the vulnerability.

SecDCP (Wang et al., 2016), a recent work, classifies processes as confidential and public. It follows an asymmetric security policy: it aims to secure only the confidential applications from a side-channel attack, and assumes that the mechanism for classification of processes is not insecure. Though it claims to provide security to the confidential process, we show in Section 3.1 that it is still vulnerable to the Flush+Reload attack.

Prior work like COTSknight (Yao et al., 2019) and DAWG (Kiriansky et al., 2018) also address similar security concerns. However, unlike PASS-P, both of these require software and OS support and incur higher performance penalties. COTSknight makes novel use of cache monitoring technology (CMT) and cache allocation technology (CAT) features of modern processors to identify and isolate suspiciously behaving processes. However, it does not consider Flush+Reload attacks. Compared to an insecure LRU baseline, COTSknight shows a slowdown of up to 5%. DAWG proposes a generic mechanism for secure way partitioning to isolate cache accesses and metadata. Compared to an approximate LRU baseline, DAWG exhibits slowdown between 0% and 15% for different experiments. PASS-P, on the other hand, shows an average slowdown of 0.35% and a maximum slowdown of 2.2% compared to insecure UCP baseline. Considering that UCP gives a 10.96% higher performance on average compared to LRU (Qureshi and Patt, 2006), we expect PASS-P to also perform favorably when augmented to UCP.

NoMo (Domnitser et al., 2012b) is an L1-cache security system which presents a performance-security tradeoff that can be tuned. Like PASS-P, it requires no software support and requires only simple changes to existing cache replacement logic. However, the NoMo configuration which gives complete security is identical to static partitioning and may degrade performance. Compared to an LRU baseline, this configuration gives a performance degradation of up to 5% and 1.2% on average.

Our goal is to devise a method that is completely secure like static partitioning, yet achieves the performance offered by dynamic partitioning schemes.

3 VULNERABILITY IN DYNAMIC PARTITIONING

Dynamic cache partitioning (DCP) protocols are runtime algorithms that dynamically distribute cache lines amongst running processes. UCP (Qureshi and Patt, 2006), for example, periodically partitions the cache lines in each set in order to maximize the total utility of caches for all running processes. The guiding principle behind UCP is that the process that has a higher 'utility' for cache lines should be allotted more number of cache lines in each set. Utility of a cache line for a process is defined as the increase in cache hit-rate if the process was given an additional cache line. At the beginning of each phase (1 million cycles in our study), UCP computes the optimum partitioning based on the utility behaviour of the processes in the previous phase and re-partitions the cache sets. To enforce the new partitioning, UCP may need to reallocate cache lines amongst processes. Previous DCP protocols including UCP are designed such that if some lines are reallocated from a process P1 to a process P2, then P1 can *still* access those lines until P2 overwrites them (Wang et al., 2016). This was done in order to avoid unnecessary cache misses on reallocated lines in each phase. *The prime reason for a side-channel attack on UCP, or in general on any DCP scheme, is the reallocation of cache lines from one process to another.*

3.1 Threat Model

Dynamic cache partitioning schemes can be vulnerable to Flush+Reload (Yarom and Falkner, 2014) and Prime+Probe (Liu et al., 2015) attacks, especially in cases where an attacker application can influence the cache partitioning decisions. In UCP, for example, an attacker program can artificially increase or decrease its utility to cause reallocation of cache lines to and from itself respectively. Moreover, to mount these attacks, the attacker process does not need any elevated privileges. The mechanism of the Flush+Reload attack is described ahead and shown in Figure 2.

1. *Flush*: The attacker takes all but one cache lines of every set by increasing its utility and flushes them as shown in step (a) in Figure 2.
2. *Execute*: The attacker returns all the flushed lines to the victim by decreasing its utility. It then waits for the victim to execute as shown in steps (b) & (c).
3. *Reload*: Attacker takes all but one lines by increasing its utility again and reloads addresses of interest as shown in step (d). A cache hit or miss on these addresses is indicative of the victim's memory accesses.

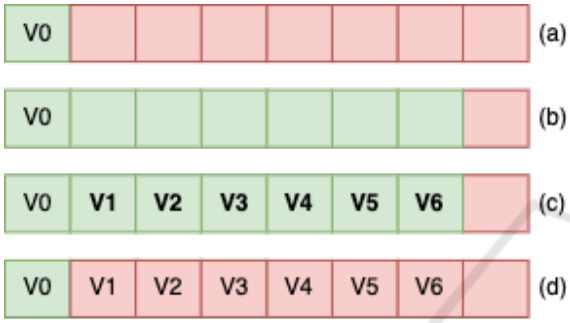

Figure 2: Mechanism for Flush+Reload attack. (a) Attacker takes lines and flushes them. (b) Attacker returns these lines. (c) Victim program executes. (d) Attacker takes these lines back and reloads targeted addresses.

Similarly, the following steps show how an attacker could mount the Prime+Probe attack. It is also shown pictorially in Figure 3.

Figure 3: Mechanism for Prime+Probe attack. (a) Attacker takes lines and primes them. (b) Attacker returns these lines. (c) Victim program executes. (d) Attacker takes these lines back and probes addresses A1 through A6.

1. *Prime*: The attacker takes all but one cache lines of every set by increasing its utility and primes them with its own data as shown in step (a) in the Figure 3.
2. *Execute*: The attacker returns all the flushed lines to the victim by decreasing its utility. It then waits for the victim to execute as shown in steps (b) & (c) in Figure 3.
3. *Probe*: Attacker takes all but one lines by increasing its utility and reloads the addresses that were previously primed as shown in step (d). A cache hit or miss on these addresses is indicative of the victim's memory accesses.

While the attacker and victim must share the code library for Flush+Reload to be mounted, there is no such requirement for Prime+Probe. For Flush+Reload, this ensures that the attacker is able to get a cache hit in the *Reload* step for addresses fetched by the victim in the *Execute* step.

Note that in most dynamic partitioning protocols, no process is permitted to possess all cache lines of a set, in order to prevent starvation of the other processes. Despite this, the attacker can extract critical information from the victim, especially over multiple iterations of the attack.

To facilitate a more granular analysis of victim's memory accesses, it is typical for the attacker to use well-known methods to slow down victim's execution considerably. For example, as described in (Gullasch et al., 2011) an attacker can achieve this by mounting a denial of service (DoS) attack on the completely fair scheduler (CFS) that is used in Linux to divide CPU time amongst running processes.

SecDCP (Wang et al., 2016), a recent work, classifies processes as confidential and public. It aims to secure only the confidential applications from a side-channel attack. Though it claims to provide security to the confidential process, we show that it is still vulnerable to the Flush+Reload attack. SecDCP only invalidates the lines that are reallocated from a public application to a confidential application, if and only if they were fetched by the public application. The lines which are taken back by the public attacker application in *Reload* step are not invalidated.

Therefore, the attacker can infer about the victim's accesses, thus making SecDCP insecure. Moreover, the partitioning decisions made by SecDCP do not factor in the demand of the confidential application, thus leading to a sub-optimal partitioning and consequent performance drop.

4 PASS-P

Performance and Security Sensitive Partitioning (PASS-P), invalidates cache lines to secure the LLC, as described in Section 4.1. Section 4.2 then describes the Modified-LRU reallocation policy that is adopted by PASS-P for an improvement in performance.

4.1 Security with Invalidation

To mitigate the side-channel vulnerability described in Section 3, the attacker must be prevented from successfully performing differential timing analysis on the reallocated lines. To stop the access of shared resources of other program, PASS-P invalidates all cache lines that are reallocated from one process to another. Because of this preemptive invalidation of lines, no process is able to cause eviction of lines of any other process. Side-channel attacks cannot be mounted in such a system, for the reasons described below.

1. *Flush+Reload*: All lines reallocated to the attacker after the *Execute* step are invalidated and the attacker gets a miss for every targeted address in *Reload* step.
2. *Prime+Probe*: The lines primed by the attacker in the *Prime* step are invalidated when they are reallocated to the victim. Hence, in the *Probe* step the attacker will get a cache miss for all these invalidated lines.

The attacker's differential timing analysis fails because all addresses that the attacker attempts to fetch result in the same cache behavior.

4.2 Modified-LRU Reallocation Policy for PASS-P

The invalidation of the cache lines in PASS-P results in a performance loss. We identify two reasons for this:

1. In UCP, when a process P1 gives up some lines of the shared LLC to another process P2, it can still access the cache lines until P2 overwrites them with its data. However, in PASS-P, due to invalidation of all reallocated lines, P1 will incur additional cache misses. As invalidation is critical for security, the performance drop is inevitable. We propose a modification in the LRU (Least recently used) reallocation policy to address the second reason (given below) and regain most of the lost performance.
2. Our experiments show that 32% of all reallocated lines are *dirty* in nature. These must be written to the main memory before their invalidation in the LLC. We observe that this invalidation can lead to a surge in memory traffic at the start of each UCP phase, as many lines may have to be written back at once. Hence, the running processes face additional delays while handling any new cache misses.

UCP uses the conventional LRU policy to choose the lines belonging to one process that should be reallocated to other processes. The LRU policy does not adequately address the above causes of poor performance. To ameliorate the effects of the second reason, PASS-P employs the Modified LRU reallocation policy.

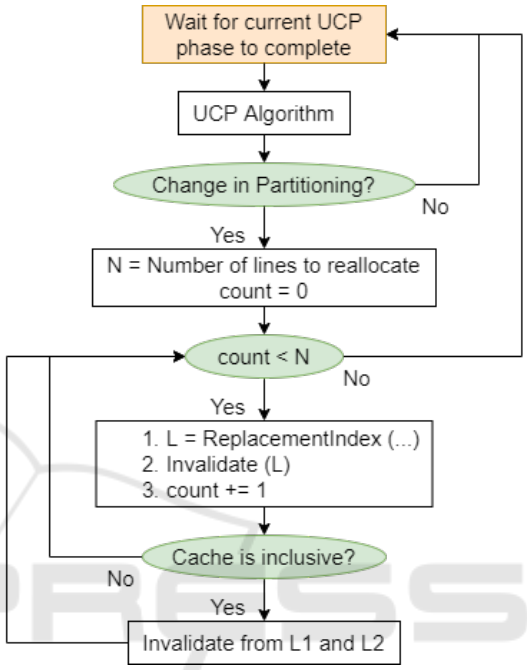

Figure 4: Flow Diagram of PASS-P's Reallocation Policy.

According to this policy, PASS-P preferentially reallocates *clean* lines over *dirty* lines from a set, if the *clean* lines are not recently used, so as to still respect the recency order. This reallocation policy reduces the number of writebacks to the main memory. We define a 'threshold fraction' $f \in [0,1]$. Our reallocation policy is defined as: *Reallocate LRU-Clean line from a set if one exists and only if the clean line is in the f fraction of the least recently used lines allocated to the process, else reallocate the (dirty) LRU line*. For instance, consider that a process has to choose a line for reallocation to another process among its 8 lines in a cache set, given $f = 0.75$. The Modified-LRU policy will inspect the 6 $(= 8 * f)$ least recently used lines and reallocate the least recently used clean line amongst them. If all of these 6 lines are dirty, our policy simply reallocates the least recently used line.

Algorithm 1 describes the selection of lines for reallocation and the entire replacement policy is shown in Figure 4. In this figure, 'N' denotes the number of cache lines to be reallocated at the end of the UCP phase, as determined by the partitioning algorithm.

Algorithm 1: Choosing a line for reallocation.

Function *ReplacementIndex* (***In***: *List L* $<$ *blockIndex, dirtybit* $>$, *f, associativity n;* ***Out***: *blockIndex*)

l = *getIndexLRUCleanLine*(L) **if** $(l != null \;\&\&$ $l \leq f * n)$ **then**
  return l
else
  return 0; *//LRU line*

Note : *getIndexLRUCleanLine*(L) *gives index of LRU clean line*

To find the best value of f, we evaluated the performance of a few pairs of memory intensive benchmarks for different values of f. Figure 5 compares the geometric mean of speedup obtained with respect to static partitioning for these different values of f.

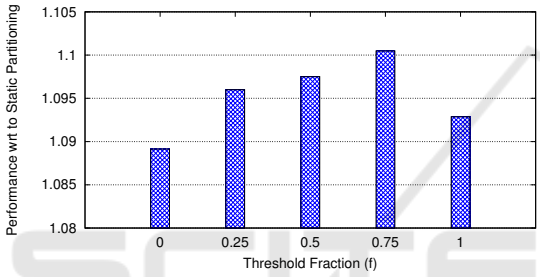

Figure 5: Geometric means of speedups of PASS-P for memory-intensive benchmark pairs with respect to static partitioning for different values of f. (For Configuration 1 in Table 1).

This graph shows that $f = 0.75$ performs the best, giving the highest speedup of 10%. The value of the $f = 0$ corresponds to the standard LRU reallocation policy, resembling the one conventionally used by UCP. The graph clearly shows that the LRU policy is not well-suited for PASS-P. $f = 1$ indicates a policy that always reallocates clean lines even if it is at MRU (Most recently used) position whenever such a clean line exists.

5 EVALUATION

We examine the performance of two benchmarks running simultaneously on two separate cores. We evaluate the performance of the two different configurations shown in Table 1 using the cycle-accurate Sniper simulator. In each experiment, two benchmarks run simultaneously on two separate cores. The PASS-P and UCP algorithms run with a phase length of 1 million cycles.

We measure performance using 'weighted speedup' metric that is the most appropriate for such multi-core systems (Qureshi and Patt, 2006).

$$WeightedSpeedup = \sum_{i^{th}\,process} (IPC_i / SingleIPC_i) \quad (1)$$

where IPC_i is the IPC of the i^{th} process in the multi-process system, and $SingleIPC_i$ is its IPC when run independently on a single core.

We show the results for twenty-five pairs of benchmarks selected from the diverse set present in the SPEC CPU2006 benchmark suite. In the first eighteen pairs, both benchmarks are memory-intensive (Navarro-Torres et al., 2019) ('MM' pairs). In the remaining seven pairs, the first benchmark is memory-intensive, while the second is compute-intensive ('MC' pairs) (Navarro-Torres et al., 2019). MM pairs are of special interest to us, because both benchmarks contend aggressively for cache lines and the optimum partitioning for UCP changes more frequently. Hence, higher number of re-allocations and invalidations take place in the course of their execution, posing a bigger challenge for the performance of PASS-P. Since compute-intensive benchmarks are not sensitive to the cache replacement policies, it is not insightful to study pairs with both compute-intensive benchmarks. In our experiments, we observed that the percentage of reallocated lines which were dirty dropped to around 22% for PASS-P in comparison to UCP's 32%.

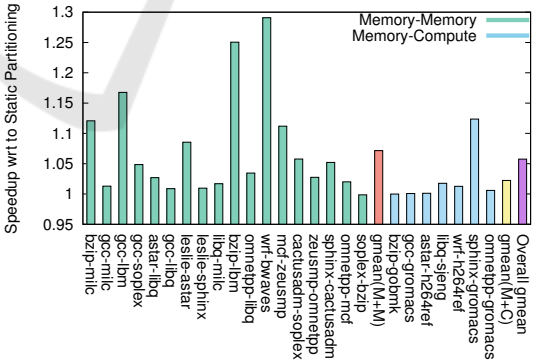

Figure 6: Comparison of Speedup of PASS-P (f=0.75) normalized to Speedup of Static partitioning for different benchmark pairs for Configuration 1 of Table 1.

Figure 6 shows the performance gain of PASS-P with respect to static partitioning. Our method with L3 as LLC gives a considerable performance gain of up to 29% and 7.2% on average for MM pairs of benchmarks. The overall average performance gain for all type of combinations is 5.8%. The lower

Table 1: Core configurations.

	Configuration 1	Configuration 2
Last Level Cache (LLC)	L3	L2
LLC size, associativity	4 MB, 16 way	256 KB, 8 way

speedup for the combination of MC pairs of benchmarks is expected, because compute-intensive programs do not have a high utility of the cache. The choice of the cache partitioning protocol does not affect compute-intensive programs' performance as much.

For 'Configuration 2' mentioned in Table 1, the performance gain is up to 33.4% and 10.6% on average as shown in Figure 7. The L2 cache, which has lower associativity (compared to L3 cache), benefits more from the proposed PASS-P policy. The higher gain in this case is because of more efficient utilization of the cache sets.

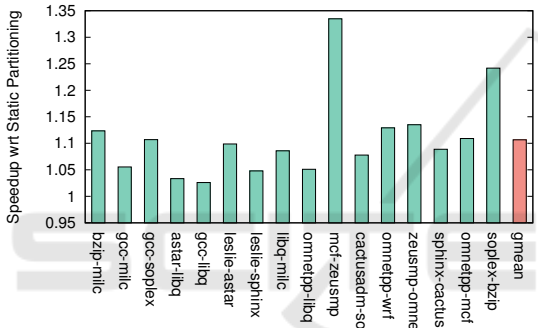

Figure 7: Comparison of Speedup of PASS-P (f=0.75) normalized to Speedup of Static partitioning for different benchmark pairs for Configuration 2 of Table 1.

Figure 8 shows the performance of PASS-P (with $f = 0.75$) for all benchmark pairs with respect to UCP. There is a performance drop of 0.50% in case of the eighteen MM pairs, while the value is even lower at 0.12% for the seven MC pairs. Overall geometric mean value for the performance drop is 0.35%. Thus, *PASS-P has only a marginal drop in performance with respect to UCP.*

6 CONCLUSION

Through this work, we have shown that side-channel attacks like Flush+Reload and Prime+Probe can be mounted on dynamic cache partitioning (DCP) protocols. While static partitioning can mitigate these attacks, it has a huge performance penalty. Through cache line invalidation and the Modified-LRU reallocation policy, we are able to overcome the security vulnerability in DCP protocols like utility-based cache partitioning (UCP), while gaining a speedup of up to 29% and on average 7.12% compared to static partitioning. We also show that this technique has a marginal performance cost of only 0.35% with respect to UCP on average. PASS-P can be applied on shared levels of cache for all dynamic partitioning protocols. Thus, it is an effective method for mitigation of side channel attacks. Extension of PASS-P to provide security against newer attacks like Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2019) is left for future work.

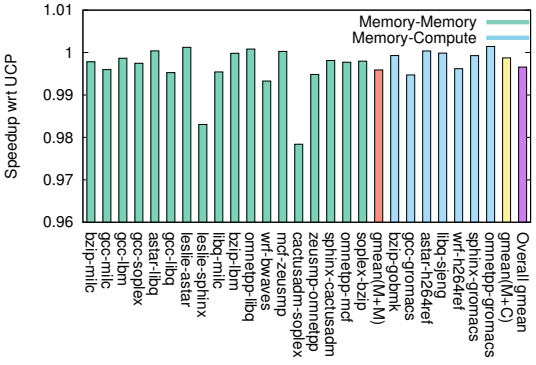

Figure 8: Comparison of Speedup of PASS-P ($f = 0.75$) normalized to Speedup of UCP for different benchmark pairs for Configuration 1 of Table 1.

ACKNOWLEDGEMENTS

We are thankful to Indo-Japanese (DST-JST) Joint Lab Grant on Intelligent CPS for supporting this project. This work has also been supported by National Security Council Secretariat (NSCS), Govt. of India. We are also thankful to Visvesvaraya PhD Scheme, Ministry of Electronics and Information Technology, Government of India for their support. We are thankful to Prof Bernard Menezes, CADSL members, ISRDC (Information Security Research & Development Center) members for their valuable suggestions.

REFERENCES

Ashokkumar, C., Giri, R. P., and Menezes, B. (2016). Highly efficient algorithms for aes key retrieval in

cache access attacks. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 261–275. IEEE.

Bernstein, D. J. (2005). Cache-timing attacks on aes.

Boran, N. K., Pinto, K., and Menezes, B. (2021). On disabling prefetcher to amplify cache side channels. In *2021 25th International Symposium on VLSI Design and Test (VDAT)*, pages 1–6. IEEE.

Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. (2012a). Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21.

Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. (2012b). Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21.

Gullasch, D., Bangerter, E., and Krenn, S. (2011). Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE.

Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., and Emer, J. (2018). Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2019). Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE.

Kong, J., Aciicmez, O., Seifert, J.-P., and Zhou, H. (2008). Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, pages 25–34.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al. (2018). Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990.

Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. (2015). Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE.

Navarro-Torres, A., Alastruey-Benedé, J., Ibáñez-Marín, P., and Viñals-Yúfera, V. (2019). Memory hierarchy characterization of spec cpu2006 and spec cpu2017 on the intel xeon skylake-sp. *Plos one*, 14(8):e0220135.

Page, D. (2005). Partitioned cache architecture as a ėide-channel defence mechanism.

Percival, C. (2005). Cache missing for fun and profit.

Qureshi, M. (Oct 2018). Ceaser: Mitigating conflit-based cache attacks via encryptedaddress and remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.

Qureshi, M. K. and Patt, Y. N. (2006). Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE.

Sanchez, D. and Kozyrakis, C. (2012). Scalable and efficient fine-grained cache partitioning with vantage. *IEEE Micro*, 32(3):26–37.

Tromer, E., Osvik, D. A., and Shamir, A. (2010). Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71.

Wang, Y., Ferraiuolo, A., Zhang, D., Myers, A. C., and Suh, G. E. (2016). Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6.

Wang, Z. and Lee, R. B. (2007). New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505.

Xie, Y. and Loh, G. H. (2009). Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, 37(3):174–183.

Yan, M., Shalabi, Y., and Torrellas, J. (2016). Replayconfusion: detecting cache-based covert channel attacks using record and replay. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE.

Yao, F., Fang, H., Doroslovački, M., and Venkataramani, G. (2019). Cotsknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 121–130. IEEE.

Yarom, Y. and Falkner, K. (2014). Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732.