SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection

Yao Wang
Cornell University
yao@csl.cornell.edu

Andrew Ferraiuolo
Cornell University
andrew@csl.cornell.edu

Danfeng Zhang
Penn State University
zhang@cse.psu.edu

Andrew C. Myers
Cornell University
andru@cs.cornell.edu

G. Edward Suh
Cornell University
suh@csl.cornell.edu

ABSTRACT

In today's multicore processors, the last-level cache is often shared by multiple concurrently running processes to make efficient use of hardware resources. However, previous studies have shown that a shared cache is vulnerable to timing channel attacks that leak confidential information from one process to another. Static cache partitioning can eliminate the cache timing channels but incurs significant performance overhead. In this paper, we propose Secure Dynamic Cache Partitioning (SecDCP), a partitioning technique that defeats cache timing channel attacks. The SecDCP scheme changes the size of cache partitions at run time for better performance while preventing insecure information leakage between processes. For cache-sensitive multiprogram workloads, our experimental results show that SecDCP improves performance by up to 43% and by an average of 12.5% over static cache partitioning.

1. INTRODUCTION

Multicore processors are widely used in today's computing systems, including cloud servers and mobile devices. In the architecture of multicore processors, the last-level cache is often shared by multiple cores to improve hardware efficiency. Each core can run a different application and contends for the cache resource.

Since the last-level cache is shared, different applications can interfere with each other by evicting each other's cache entries. Because cache misses take much longer than cache hits, an application can thus affect the execution time of another application, which may be exploited to initiate timing channel attacks. Indeed, previous work [2, 11, 8] has shown various timing channel attacks based on cache interference. It is important to defend against these timing channel attacks because they introduce serious security threats in modern computing systems. In cloud computing, a user's applications are scheduled to run concurrently with other applications, which could be malicious and try to infer private information about the user through cache timing channel attacks. A similar problem exists in current mobile devices such as smartphones. If a user downloads an application that contains malware, this malicious application might run concurrently with the user's banking application, stealing confidential information.

Previous work [10] proposes static cache partitioning to defend against cache timing channel attacks. In static cache partitioning, the shared cache is statically divided into multiple partitions. Each application can only use its own partition of the cache, thus effectively eliminating any cache interference. However, static cache partitioning incurs significant performance overhead because the partition size cannot adapt to the changing cache behavior of each application. To improve the performance of static cache partitioning, previous work [14, 12, 17, 13] proposes dynamic cache partitioning techniques that change the partition size dynamically based on the run-time demands of applications. Unfortunately, previous dynamic partitioning techniques are still vulnerable to cache timing channel attacks because the partitioning policy relies on the run-time behavior of each application. Consequently, a malicious application can learn confidential information about another application by observing how cache partition sizes change.

In this paper, we present Secure Dynamic Cache Partitioning (SecDCP), a new scheme that dynamically partitions the cache yet does not leak confidential information through timing channels. We note that applications running on a multicore processor have diverse security requirements. Public applications (e.g., simple games) do not contain confidential information, hence they require little protection against information leakage. On the other hand, confidential applications (e.g., medical app), which involve computing on confidential data, need strict timing channel protection. SecDCP leverages this asymmetry of applications' security requirements to perform dynamic cache partitioning while satisfying each application's security needs. In other words, SecDCP prevents information leakage from confidential applications to public applications, but allows information to flow from public applications to confidential applications. We call this kind of timing channel protection as **one-way protection**.

Using one-way protection, SecDCP is able to use the cache demand of public applications to allocate the partition sizes at run time. Specifically, SecDCP grants public applications larger cache partitions when these applications can benefit significantly from more cache. Conversely, SecDCP shrinks the partition size of public applications when they cannot effectively utilize their current partitions, allowing confidential applications to improve their performance. Consequently, system performance improves as a whole even though SecDCP uses only the cache demand of public applications to change partition sizes. Our experiment results show that SecDCP outperforms static partitioning by 12.5% on average for cache-sensitive workloads.

This paper has the following main contributions:

- The paper proposes SecDCP, a secure dynamic cache partitioning scheme that defeats timing-channel attacks caused by interference between processes, including both side-channel attacks and covert-channel attacks.
- The paper discusses extending SecDCP to support a general security policy and shows the extension is non-trivial because of new security threats.

DAC '16, June 05-09, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897937.2898086

- This paper evaluates the proposed schemes and shows that SecDCP improves performance even though dynamic partitioning is based solely on the cache demand of the public applications.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 discusses our threat model. Section 4 describes details of the secure dynamic partitioning scheme. Section 5 evaluates the proposed scheme. Section 6 concludes the paper.

2. RELATED WORK

Recent studies [2, 11, 8] have shown that shared caches are vulnerable to timing channel attacks. In general, cache timing channel attacks exploit the cache interference between the victim and the attacker. For example, a victim process may evict some of the attacker's cache lines, delaying the attacker process. The attacker process can then use this timing information to infer confidential information (e.g., encryption keys) about the victim process.

Software solutions [9] rely on rewriting the software to remove known timing channels, hence they are attack-specific and incur significant performance overhead. Hardware solutions provide more general protection against timing channel attacks. Various secure cache designs [10, 15, 16, 7] haven been proposed recently. The approaches taken by these designs fall into two categories: randomization and partitioning. In the randomization approach [15, 16, 7], the memory-to-cache mapping is randomized to obfuscate the attacker's measurement. Although this approach can hide which cache line has been accessed by the victim, randomization does not hide the number of cache accesses. Hence, randomization cannot defend against covert channel attacks, in which an attacker intentionally manipulates the number of cache accesses to send confidential information. In the partitioning approach [10], the cache is statically divided into multiple partitions. Each partition can only be used by one process, thereby eliminating the cache interference between the victim and the attacker. However, static cache partitioning incurs high performance overhead because the partition size cannot adapt to the runtime cache behavior of each process.

Previous work [14, 12, 17, 13] has studied dynamic cache partitioning techniques, utilize the runtime cache demand information of each application to adjust the partition sizes. However, previous dynamic partitioning schemes are vulnerable to timing channel attacks because the dynamic partition sizes reveal the runtime demand of confidential applications. To the best of our knowledge, this work is the first to propose dynamic cache partitioning as a security mechanism to defend against cache timing channel attacks.

3. THREAT MODEL

In this paper, we aim to remove cache timing channels between different applications in a multicore processor. The processor's memory hierarchy consists of one or more levels of private caches and a shared last-level cache. If left unmanaged, the shared last-level cache can be vulnerable to timing-channel attacks.

A security class is assigned to each application by the underlying OS, which might make these security class assignments based on user preferences. We assume the mechanism for assigning security classes is secure and cannot be compromised by the attacker. The security classes form a hierarchical security policy, as shown in Figure 1. There are N security tiers, and each tier contains an arbitrary number of security classes. The security policy is defined by the following rule:

- Information is allowed to flow from security class A to security class B if and only if $A \in$ Tier i, $B \in$ Tier j and $i < j$.

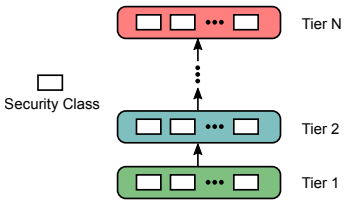

Figure 1: Hierarchical security policy.

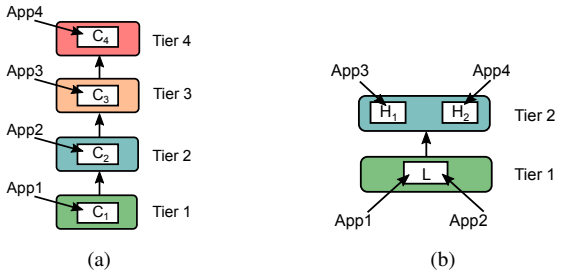

Figure 2: Common security policies.

The security policy has two implications. First, information may not flow between two security classes in the same security tier. This is useful for the use case where multiple mutually distrusting confidential applications (e.g., banking, medical) are running concurrently. Second, information *is* allowed to flow from a lower security tier to a higher security tier. This represents the use case where public applications (e.g., web browsing, search engine) are running together with confidential applications. Information is allowed to flow from public applications to confidential applications, but not the other way around.

The security policy is general enough to support common use cases. For example, a government or military database system usually employs the multilevel security (MLS) policy, which consists of four hierarchical levels. This MLS policy can be represented in our security policy by four security tiers with each tier containing one security class, as shown in Figure 2a. Our security policy can also represent the mobile scenario with two security tiers, as shown in Figure 2b. Tier 1 only contains one security class L, which is assigned to public applications; recall that a security class is not the same as an application. Multiple applications can share a single security class as long as they do not require protection against each other. Tier 2 contains 2 security classes, with each security class assigned to a different confidential application.

Given that the security policy defines which information flows are allowed between security classes, the goal is to prevent disallowed information flows through cache timing channels. Specifically, SecDCP prevents timing channel attacks that exploit cache interference between processes, including prime-probe attacks [11, 9], evict-time attacks [9] and flush-reload attacks [5]. SecDCP does not consider timing channel attacks based on interference within a process, such as cache collision attacks [4].

SecDCP assumes a strong attacker model. The attacker is able to measure the timing of its own individual cache accesses. In addition to side channel attacks, SecDCP also deals with covert channel attacks, in which a compromised application in a high security tier tries to leak information to another application in a low security tier. SecDCP assumes hardware counters are controlled by the OS or hypervisor and are not accessible by user processes. Hence, the attacker cannot infer the victim's cache behavior by directly accessing cache hardware counters. SecDCP does not consider physical side channel attacks through power [6] or sound [1].

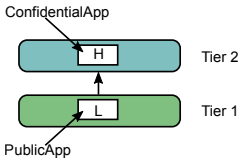

Figure 3: Two security classes.

4. SECDCP DESIGN

4.1 Challenge

Previous dynamic cache partitioning techniques are vulnerable to cache timing channel attacks, as follows.

Partition size depends on the demand of confidential applications. To decide the partition size, previous dynamic cache partitioning schemes consider the cache demands of all processes. This means the partition sizes can reveal the cache demand of each process. By observing the change in its own partition size, a public application is able to infer the run-time demand of a confidential application.

Changing partition size is not strictly enforced. When changing the cache partition size, previous dynamic cache partitioning methods [12, 17, 13] enforce the new partition size at replacement time. If the size of a cache partition decreases, its cache lines are not immediately evicted. Instead, they remain in the cache until some cache lines from another process replace them. This means the eviction of one process's cache lines depends on other processes' cache accesses—a vulnerability for timing channel attacks.

Since dynamic cache partitioning is vulnerable to timing channel attacks, prior secure partitioned cache designs have used static cache partitioning, incurring significant performance overhead. A natural question is whether we can achieve the benefits of both schemes. In this paper, we show that secure dynamic cache partitioning is feasible under our security policy.

We observe that the hierarchical security policy is inherently asymmetric. We only need to prevent information flow from confidential applications to public applications. Taking advantage of this asymmetry, we propose Secure Dynamic Cache Partitioning (SecDCP), which significantly improves the performance of static cache partitioning while meeting the security requirements.

To illustrate the ideas of SecDCP, we will first describe SecDCP design for a simple hierarchy with two security classes.

4.2 SecDCP for Two Security Classes

Consider the security policy in Figure 3. We assign security class H to the confidential application and security class L to the public application. The goal is to prevent information flow from H to L through cache timing channels. To achieve this goal, SecDCP follows a couple of design rules: 1) Partition size is independent of confidential applications. 2) Operations for changing partition size leak no information about confidential applications.

We divide SecDCP into two parts: *Partition Allocation Algorithm (PAA)* and *Partition Enforcement Mechanism (PEM)*. *PAA* is responsible for allocating the size of each partition at run time. *PEM* is responsible for enforcing the new partition size after *PAA* picks a new cache allocation.

4.2.1 Partition Allocation Algorithm (PAA)

Unlike previous dynamic cache partitioning techniques in which the cache allocation is dependent on the demands of both L and H, *PAA* only uses the demand from L to allocate the partition size. We divide the time into epochs of length T. At the end of each epoch,

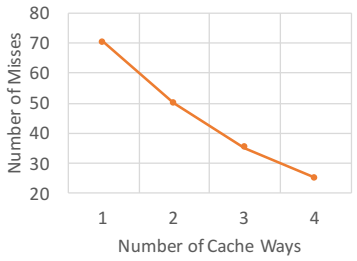

Figure 4: Generated miss curve by UMON.

PAA allocates a new partition size based on the demand of L. Inside each epoch, we use utility monitors (UMON) [12] to collect the utility information of L. UMON generates miss curves similar to Figure 4. The miss curve is a function that maps the number of cache ways to the number of cache misses. Using the miss curve, we can calculate the gain for increasing the cache size as well as the loss for decreasing the cache size. Assume the total number of cache misses for L in an epoch is N and the current partition size for L is X ways, we define the gain and loss as follows:

$$gain = [MISS(X) - MISS(X+1)]/N \quad (1)$$

$$loss = [MISS(X-1) - MISS(X)]/N \quad (2)$$

MISS(X) means the number of cache misses with X cache ways. Gain indicates the percentage of cache misses for L that can be reduced when we increase L's partition size by one cache way. Loss indicates the percentage of additional cache misses for L that will be introduced when we decrease L's partition size by one cache way.

The gain and loss parameters are fed into *PAA*. *PAA* defines two threshold values, th_{inc} and th_{dec}, to determine the new partition size. If the gain is larger than th_{inc}, L's partition size increases by one way. Otherwise, if the loss is smaller than th_{dec}, L's partition size decreases by one way.

Note that *PAA* only considers the demand of L when allocating the partition size. This design choice is made to prevent information leakage from H to L. When L increases its partition size, H's partition size decreases accordingly. Although H may suffer from the reduced partition size, overall system performance can still improve since L will get significant improvement (higher than th_{inc}). To prevent unfairness and starvation, our default design always reserves one cache way for H. On the other hand, when L decreases its partition size, H's partition size increases accordingly. This creates an opportunity for H to improve its performance with more cache ways, while the performance drop of the L partition is bounded (less than th_{dec}).

4.2.2 Partition Enforcement Mechanism (PEM)

After *PAA* allocates a new partition size, PEM enforces the new allocation in a way that leaks no information from H to L. A naive approach is to flush the entire cache way whenever a way is reallocated to another security class. But flushing can introduce significant performance overhead. PEM improves the performance by treating increases and decreases to partition size differently. We associate each cache line with a 1-bit identifier (*ID*) to indicate which security class fetched the cache line. When L's partition size decreases, PEM checks the *ID* and flushes all the cache lines that belong to L in the adjusted cache way before reallocating it to H. However, when L's partition size increases, PEM does not flush the cache way that gets reallocated from H to L. As a result, H's cache

lines in this cache way can be evicted by L's cache lines, and H can learn about the cache accesses of L. However, this information flow is benign according to our security policy. The optimization improves the performance of H by allowing H to get cache hits in its cache lines in L's partition until L evicts them.

4.3 Efficiency Discussion

SecDCP dynamically partitions the cache at run time by utilizing cache demand information from the public application, while previous dynamic cache partitioning schemes such as utility-based partitioning [12] use information from all applications to decide partition sizes. How can SecDCP make good partitioning decisions when it only sees information from one side?

It is known that applications show phase behavior. To simplify analysis, we categorize application phases into cache-sensitive and cache-insensitive phases. Assume a confidential application is running concurrently with a public application. The state of concurrent application phases, (PublicApp, ConfidentialApp), can be divided into four cases: (I, I), (I, S), (S, I), (S, S). Here I represents a cache-insensitive phase and S a cache-sensitive phase. We analyze performance case-by-case:

- (I, I): The cache partition size does not affect performance much, so SecDCP and utility-based partitioning achieve similar performance.
- (I, S): Both schemes allocate most of the cache to the confidential application because the public application does not need cache.
- (S, I): Utility-based partitioning will allocate most of the cache to the public application. SecDCP's allocation will achieve the same outcome if the threshold is properly chosen.
- (S, S): Utility-based partitioning will find an allocation that benefits both applications the most, whereas SecDCP's allocation is dependent on the threshold value.

In a summary, SecDCP achieves almost the same performance as utility-based partitioning when the public application is cache-insensitive. For the other two cases, the performance of SecDCP relies on the accuracy of a threshold value. In our experiments, SecDCP uses the same threshold value (20%) across different benchmarks and achieves similar performance as utility-based partitioning. Our analysis and experimental results indicate that using information from one side is enough for dynamic partitioning in many cases.

4.4 SecDCP for General Case

We now describe SecDCP for a more general security policy as shown in Figure 1, where there are multiple security tiers. It may seem straightforward to extend SecDCP for a general security policy, but this extension turns out to be non-trivial.

4.4.1 Partition Allocation Algorithm (PAA)

For a general security policy, *PAA* uses UMON to track the cache demand of each security class separately. Using the allocation algorithm, *PAA* is able to figure out the change in partition size for each security class. However, a new problem arises when *PAA* tries to reallocate the partition size. In a general security policy, if one security class needs to increase its partition size, it can take a cache way from any security class that is in a higher tier in the security policy. We found that security vulnerabilities exist if *PAA* is not designed carefully.

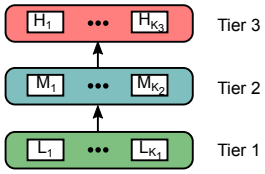

Figure 5: Security policy example.

Insecure Designs.

Without loss of generality, we discuss the insecure designs in the case of 3 security tiers, as shown in Figure 5.

Case 1: Suppose L_1 wants to increase its partition size. Meanwhile, M_1 wants to decrease its partition size and M_2 wants to keep its partition size the same. A performance oriented algorithm tends to pick a cache way from M_1 to reallocate to L_1. If M_2 knows L_1 is cache-intensive, but observes that it does not lose any cache way, it can infer M_1 may be decreasing its partition size. Hence, M_2 can learn the run-time cache demand of M_1.

Case 2: Assume M_1 wants to increase its partition size. M_1 gradually takes cache ways from H_i where ($1 <= i <= K_3$) until there is no available cache way in security tier 3. Now security class M_2 starts to request for larger partition size. However, M_2 cannot get more cache ways because tier 3 has no extra cache ways. If M_2 knows that applications in tier 1 are cache-insensitive, then M_2 can infer that a security class in tier 2 has high cache demand.

The pitfalls can be summarized as follows: 1) Dynamic arbitration between incomparable security classes based on cache demand. 2) First-Come, First-Serve (FCFS) allocation.

Secure Allocation Design.

Our secure allocation design avoids the aforementioned pitfalls. We assume a general security policy that consists of N security tiers with each tier containing K_i incomparable security classes where $1 <= i <= N$. For the convenience of description, we use the notation $C_{i,j}$ to denote the jth security class in security tier i.

If a security class $C_{i,j}$ wants to increase its partition size, *PAA* will first check security tier $(i+1)$. There are K_{i+1} security classes in tier $(i+1)$. We assume each security class has P cache ways that can be reallocated to other security classes initially. There are $K_{i+1} * P$ available cache ways for allocation in tier $(i+1)$. To avoid the FCFS allocation pitfall, *PAA* reserves certain number of cache ways for each incomparable security class. Since there are K_i incomparable security classes in tier i, each security class in tier i can get at most $\lfloor K_{i+1} * P/K_i \rfloor$ cache ways from tier $(i+1)$. Hence, *PAA* checks the reserved cache ways for security class $C_{i,j}$ in tier $(i+1)$. If there exists one cache way that is currently occupied by a security class $C_{i',j'}$ where $i' > i$, then this cache way can be reallocated to $C_{i,j}$. If multiple cache ways satisfy the condition, *PAA* picks one cache way using a strict round-robin policy, thereby avoiding the pitfall of dynamic arbitration based on cache demand. If no cache way satisfies the condition, *PAA* will then move to tier $(i+2)$ and repeats the same procedure. The algorithm traverses through the security tiers from tier $(i+1)$ to tier N, and terminates when a target cache way is found or the search reaches tier N.

If a security class $C_{i,j}$ wants to decrease its partition size, *PAA* will simply reallocates the extra cache way to one security class in tier $(i+1)$. If there are multiple security classes in tier $(i+1)$, *PAA* picks one security class using strict round-robin policy.

4.4.2 Partition Enforcement Mechanism (PEM)

PEM does not flush the cache way when the partition size increases. However, when a partition size decreases, *PEM* needs to

Core count	2 / 4
Core model	2GHz Out-of-Order, ARM ISA
L1 caches	Private, 32kB, 2-way set associative, split D/I
L2 caches	Shared, 1MB, 8-way set associative / Shared, 2MB, 16-way set associative
Memory	200-cycle latency, 8GB/s peak memory BW

Table 1: System configuration.

cache-insensitive	astar, libquantum, gobmk, h264ref, hmmer, sjeng
cache-sensitive	bzip2, mcf, soplex, xalan

Table 2: Program categorization.

carefully flush specific cache lines to avoid timing channel attacks. Assuming a security class $C_{i,j}$ gives up one cache way to be reallocated to another security class $C_{i+1,j'}$, *PEM* will check each cache line in the reallocated cache way. If the security class of a cache line is incomparable with or lower than $C_{i+1,j'}$, this cache line needs to be flushed.

5. EVALUATION

5.1 Experimental Methodology

We use an architecture simulator, gem5[3], to evaluate the performance of SecDCP. We model a multicore system that is configured with private L1 caches and a unified shared L2 cache, as shown in Table 1. We ran multiprogram workloads that consist of SPEC CPU2006 programs. We fast-forward the simulation until every program in the workload has reached 1 billion instructions, after which the simulator starts detailed timing simulation. The simulation terminates when every program has run at least 250 million instructions.

We use 10 SPEC CPU2006 programs to form our multiprogram workloads. We use profiling to categorize these programs [12], as shown in Table 2. Cache-insensitive programs do not benefit significantly from cache size increase because they have few cache accesses or because their working set fits in a small cache size. In contrast, cache-sensitive programs can continuously benefit from increasing cache size up to the entire cache. For convenience, we mark cache-sensitive programs with the letter *S* and mark cache-insensitive programs with the letter *I*. We use weighted speedup as the metric to evaluate the performance. For a system with programs running concurrently, weighted speedup is defined as the sum of each program's IPC normalized to the IPC when the program is running by itself:

$$Weighted\ Speedup = \Sigma(IPC_i/SingleIPC_i) \quad (3)$$

5.2 Performance

We first study the performance of SecDCP when there are two security classes. We use the naming convention program1_program2 for a workload, meaning that program1 belongs to *L* and program2 belongs to *H*. We compare SecDCP with three other schemes. The first scheme is static partitioning, which statically divides the cache into two equally-sized partitions. The second scheme is no partitioning where the entire cache is managed using LRU replacement. The last scheme is utility-based partitioning [12] that dynamically partitions the cache using the cache demand of both programs. For our SecDCP scheme, we set the threshold value, th_{inc} and th_{dec}, to be both 20% in these experiments. Out of the four schemes, static partitioning and SecDCP are secure while no partitioning and utility-based partitioning are insecure.

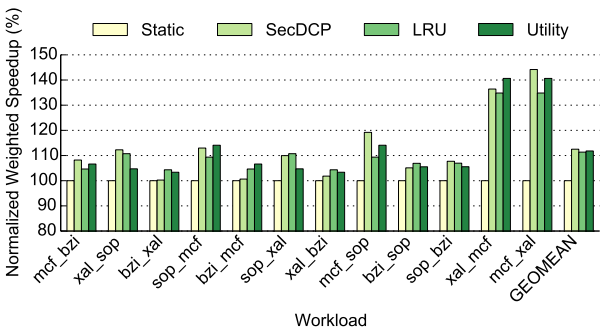

Figure 6: Performance for *SS* workloads.

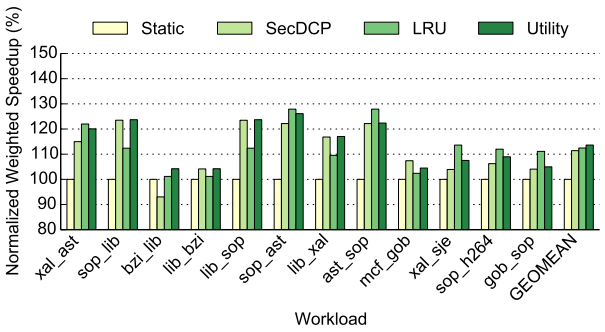

Figure 7: Performance for *SI* workloads.

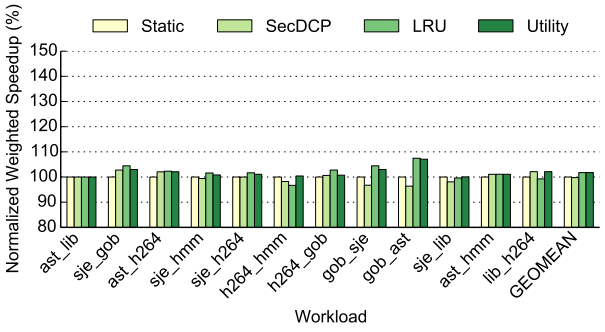

Figure 8: Performance for *II* workloads.

Figure 6 shows the results for *SS* workloads, which are mixes of two cache-sensitive programs. We normalize the weighted speedup of each scheme to that of static partitioning. For these workloads, SecDCP achieves 12.5% improvement over static partitioning on average. In some cases, the improvement can reach as much as 40%. This is because *SS* workloads are cache-sensitive, requiring efficient utilization of the limited cache space. Static partitioning incurs high performance overhead since it cannot adapt the partition size to meet the changing cache demand of each program. SecDCP increases the partition size for *L* when the *L* program is able to improve its performance with larger partition size. On the other hand, SecDCP decreases the partition size for *L* when it will not hurt the performance of the *L* program by much, thus giving more cache space to the *H* program, which can opportunistically improve its performance. On average, SecDCP achieves similar performance to no partitioning and utility-based partitioning.

Figure 7 shows the performance for *SI* workloads, which are randomly selected mixes of a cache-sensitive program and a cache-

insensitive program. On average, SecDCP achieves an 11.4% improvement over static partitioning. For *SI* workloads, SecDCP will gradually reduce the partition size of the cache-insensitive program, greatly improving the performance of the cache-sensitive program. Utility-based partitioning can also achieve the same goal, which is why we see similar speedups between SecDCP and utility-based partitioning for each workload. We also see in some cases (e.g., bzi_lib) that SecDCP performs worse than static partitioning. This is because the threshold value we choose (i.e., 20%) does not match the cache demand of *bzip*2. In this workload, *bzip*2 keeps decreasing its partition size because the loss (defined in equation 2) is less than 20%. However, *libquantum* is a streaming program that cannot utilize the increasing cache size. System performance decreases as a result. If we swap the programs (i.e., lib_bzi), *bzip*2 gets most of the cache because *libquantum* gives up most of its cache ways. As the figure shows, SecDCP outperforms static partitioning by 5% for lib_bzi.

Figure 8 shows the performance for *II* workloads, in which both programs are cache-insensitive. Since the performance of the program does not depend much on cache size, all schemes perform similarly.

To understand the impact of having more security classes, we also studied the performance of SecDCP with four cores. We use a 2MB, 16-way set associative L2 cache, shared by four programs. The security policy is the linear security policy shown in Figure 2a. We compare SecDCP with static partitioning. The static partitioning scheme divides the cache into four partitions, each a four-way cache. We picked 32 four-program workloads that consist of *SS*, *SI* and *II* types (two programs from each category). The results show that SecDCP achieves 6.4% improvement over static partitioning on average. For quite a few benchmarks, the improvement can reach 20%. However, we do see some workloads for which SecDCP performs worse than static partitioning. This is again due to the imprecision of the threshold value (20%).

5.3 Threshold and Security Policies

We tried different threshold values from the set {2%, 5%, 10%, 15%, 20%, 25%}. We found the threshold value can affect the performance significantly for most workloads, hence picking a good threshold is important. By comparing against the profiling results for individual programs, we found that the optimal threshold value can be estimated based on the steepness of a program's miss curve. For most workloads, a threshold value of 20% performs well, and on average is within 2.5% of the optimal threshold.

To understand the impact of security policies, we studied performance under another policy shown in Figure 2b. In this case, the static cache partitioning scheme divides the cache into 3 partitions. The partition size for L is 8 cache ways because two programs share the partition, while the partition sizes for H_1 and H_2 are both 4 cache ways. SecDCP achieves up to 21% and on average 3.3% improvement over the static partitioning. The speedup is lower than that of the linear security policy, because allowing two public applications to share a cache partition improves the performance of the baseline static partitioning.

6. CONCLUSION

We present SecDCP, a secure dynamic cache partitioning scheme that improves the performance of static cache partitioning while meeting the security requirements specified by a hierarchical security policy. SecDCP uses only the cache demand of public applications to dynamically determine the cache partition size. SecDCP then enforces the new partition size securely and efficiently by only flushing cache lines when necessary. SecDCP also supports a non-trivial extension to general security policies.

7. ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation under grant CNS-1513797 and an equipment donation from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or Intel.

8. REFERENCES

[1] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. Acoustic side-channel attacks on printers. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[2] D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[4] J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, 2006.

[5] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[6] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, 1999.

[7] F. Liu and R. B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[8] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy*, 2015.

[9] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[10] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Eprint archive*, 2005.

[11] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[12] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[13] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[14] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[15] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[16] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[17] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.