Secure Dynamic Memory Scheduling against Timing Channel Attacks

Yao Wang, Benjamin Wu, and G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
{yw438,bhw49,gs272}@cornell.edu

ABSTRACT

This paper presents SecMC, a secure memory controller that provides efficient memory scheduling with a strong quantitative security guarantee against timing channel attacks. The first variant, named SecMC-NI, eliminates timing channels while allowing a tight memory schedule by interleaving memory requests that access different banks or ranks. Experimental results show that SecMC-NI significantly (45% on average) improves the performance of the best known scheme that does not rely on restricting memory placements. To further improve the performance, the paper proposes SecMC-Bound, which enables trading-off security for performance with a quantitative information theoretic bound on information leakage. The experimental results show that allowing small information leakage can yield significant performance improvements.

1. INTRODUCTION

Today's computing systems are becoming increasingly vulnerable to timing channel attacks because hardware resources in a multi-core processor are shared among multiple security domains. In cloud computing services, a user's virtual machine (VM) may be scheduled to run on the same physical machine as an attacker's VM, which may be able to infer confidential information through timing channel attacks. In fact, a practical timing channel attack through shared data caches has already been demonstrated by Ristenpart et al. [1] on commercial Amazon EC2 servers.

Recently, Wang et al. [2] showed that timing channel attacks are feasible through a shared memory controller. By measuring the delay of its own memory accesses, an attacker is able to infer the dynamic memory demand of a concurrently running program, which may allow inferring secret information about a victim program or receiving covert messages from another attacker program. Hunger et al. [3] also found that a covert timing channel caused by accesses to main memory has a high capacity over 500Kbps, greater than the capacity of timing channels through caches and branch predictors.

Unfortunately, existing techniques to remove memory timing channels incur high overhead and/or place significant restrictions on memory allocations. Temporal Partitioning (TP) [2] shows that static turn scheduling can eliminate the memory timing channels, but incurs high performance overhead in order to avoid interference even in the worst case where requests access the same bank. Bank Triple Alternation (BTA) [4] improves the performance of TP by restricting DRAM schedules to enforce consecutive requests access different banks. However, the overhead can be still quite high for memory-intensive applications. Rank partitioning [4] can significantly improve performance by restricting memory allocations so that different security domains cannot share a rank. Yet, such strict spatial partitioning significantly restricts memory allocations and is difficult to deploy in many real-world applications. For example, in cloud computing, rank partitioning implies that a system can only support a small number of VMs, limited by the number of ranks (typically no more than 8).

In this paper, we propose new secure memory controller designs, which can significantly reduce performance overhead without restricting data placements. The first design, named SecMC-NI (NI stands for non-interference), completely eliminates timing interference among security domains while allowing the peak memory bandwidth to be the same as rank partitioning. SecMC-NI allows multiple requests to be issued within a short period by interleaving accesses to different banks and ranks. Our evaluation results show that SecMC improves the performance of BTA by 45% on average.

To further improve the performance of secure memory scheduling schemes, we explore a new dimension in the trade-off space: security. Instead of completely removing timing channels, which requires DRAM scheduling to ensure no timing dependence between security domains is possible under any traffic patterns, we show that allowing a small amount of information leak can significantly reduce overhead. The new scheme, named SecMC-Bound, hides most timing interferences by delaying memory responses, and provides an information theoretic bound on information leakage by bounding the number of cases that the interference is visible to a program. To the best of our knowledge, this work represents the first protection scheme that defends against memory timing-channels between concurrent programs on a multi-core processor while enabling a trade-off between security and performance with a quantitative security guarantee.

The rest of the paper is organized as follows. Sec-

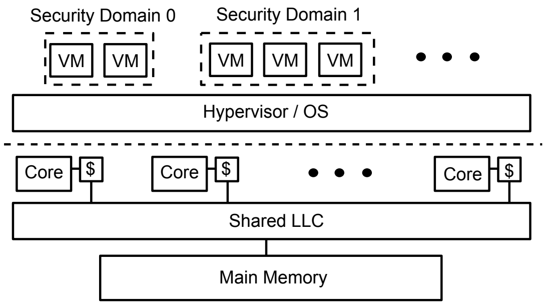


Figure 1: A multi-core example for cloud computing.

tion 2 introduces the background and our threat model. Section 3 describes the SecMC-NI scheme. Section 4 describes the SecMC-Bound scheme. Section 5 evaluates the proposed schemes. Section 6 discusses the related work and Section 7 concludes the paper.

2. THREAT MODEL AND BACKGROUND

2.1 Threat Model

In shared multi-core systems, one application can impact the delay observed by another application exploiting interference in shared resources such as a memory bus. A recent study by Wang et al.[2] showed that the interference in a shared memory controller can be exploited for timing channel attacks.

In this paper, we aim to provide timing-channel protection for shared memory controllers on a multi-core processor running multiple mutually distrusting security domains. A security domain is defined as a set of software modules (VMs, processes, threads) that do not need timing isolation among them [2]. Figure 1 shows an example for a cloud computing platform. A security domain may contain multiple VMs as long as these VMs do not require timing channel protection among them. For example, VMs belonging to the same user can be grouped into a single security domain. The hypervisor/OS is responsible for scheduling these VMs to the underlying processor and tagging each VM with the correct security domain ID. We assume the hypervisor/OS is trusted and is not compromised by the attacker.

The multi-core processor has a shared last-level cache (LLC) and a shared memory controller, both of which are vulnerable to timing channel attacks. In this paper, we focus on the protection against timing channel attacks in the memory controller. We assume that the shared LLC and the on-chip network are protected with other schemes proposed in previous studies [5, 6, 7, 8, 9, 10, 11]. The proposed timing-channel protection schemes for memory controllers are orthogonal to and compatible with existing protection schemes for other shared resources.

In memory timing channels, information may leak from one program to another in two ways. The first is through memory access rates. For example, a receiver program will observe low memory bandwidth if a sender program issues a large number of memory requests and high bandwidth when there is no request from the sender. The second is through bank or rank conflicts. Even if programs issue memory requests at a fixed rate, observed memory latencies will be different depending on whether they access the same bank/rank or not. If two programs access the same bank or rank, the effective bandwidth that they observe will be lower than when they access different banks or ranks. To be secure, a protection scheme needs to prevent both types of information leakage, controlling memory request rates as well as bank/rank conflicts.

We consider both unintentional and intentional information leaks in our threat model. Thus, we aim to prevent both side-channel and covert-channel attacks.

2.2 Existing Protection Approaches

Several techniques [2, 12, 4] have been proposed to remove timing channels in shared memory controllers. They are all based on the basic idea of time-multiplexing with a fixed schedule. Temporal Partitioning (TP) divides time into fixed-length turns and allocates turns to each security domain using a fixed schedule, typically a round-robin schedule. At the end of each turn, a dead time is added to ensure that in-flight transactions cannot interfere with requests in the following turn. During the dead time, no new request can be issued.

The time interval that can ensure no interference between consecutive memory accesses depend on which memory locations are accessed. For the DRAM parameters used in our experiments, two consecutive accesses need to be separated by at least 43 cycles when they access the same bank, 18 cycles when accessing different banks in the same rank, and 6 cycles when accessing different ranks. Without any restriction, the dead time must be long enough (43 cycles) to drain any type of request. Because of the wasted cycles during the dead time, TP incurs significant performance overhead.

To improve the performance of TP, Shafiee et al. [4] proposed Bank Triple Alternation (BTA), which divides the memory banks into three bank groups. Consecutive memory requests are restricted to always access different bank groups, which ensures that there cannot be any bank conflict between turns. This enables using short (18 cycle) turns. On the other hand, only a subset of banks can be accessed in each turn. Because of the shorter turns, BTA shows a significant speedup over TP. We compare our scheme with BTA, which is so far the best performing scheme with no restriction on memory allocation.

To further reduce performance overhead, researchers have proposed to combine spatial partitioning with temporal partitioning. In spatial partitioning, the physical memory is partitioned among different security domains. For example, bank/rank partitioning [2, 4] restricts security domains to place data in different banks or ranks. This restriction ensures that memory requests from consecutive turns access different banks/ranks. The dead time between requests thus can be shorter (6 cycles with rank partitioning). However, spatial partitioning, especially rank partitioning, seriously constrains data placement, making it difficult to deploy in practice. If the number of security domains is large, spatial partitioning may be simply infeasible due to the limited number of ranks and banks. For example, typical sys-

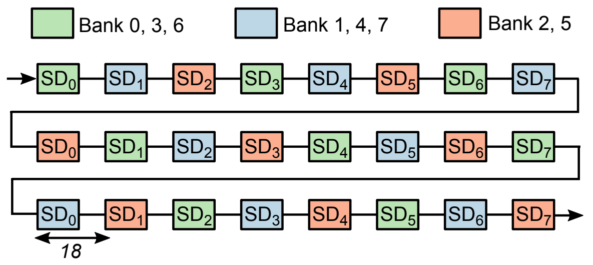


Figure 2: Bank triple alternation schedule example.

tems have no more than 8 ranks per channel. In cloud computing, rank partitioning may imply that a system cannot keep more than 8 virtual machines in memory. Spatial partitioning also leads to high memory fragmentation. VMs with a small memory footprint waste allocated memory while VMs with a large memory footprint suffer from insufficient memory.

3. SECMC-NI: EFFICIENT SCHEDULING WITH ZERO INFORMATION LEAK

3.1 Limitations of BTA

Figure 2 shows an example BTA schedule for 8 security domains and 8 banks. The 8 banks are divided into 3 bank groups, and each security domain (SD_i) can only access one of the bank groups in each turn. The schedule ensures that consecutive memory requests always access different banks, hence the time interval between them can be as short as 18 cycles using our DRAM timing parameters, much shorter than 43 cycles for TP.

However, the fixed scheduling of BTA leads to several inefficiencies. First, if two requests from the same security domain access different banks in the same bank group, the second request needs to wait for 24 (3*8) turns even though they only need to be separated by 18 cycles. Second, requests arriving at a "bad" time can be delayed significantly. For example, if a request for bank 0 from SD_0 arrives at cycle 6, it must wait for another 24 turns. Finally, BTA does not consider ranks when determining the schedule. Requests to different ranks are still separated by 18 cycles even though they only need to be separated by 6 cycles.

3.2 SecMC-NI Algorithm

The inefficiencies in BTA come from the static nature of the scheduling algorithm; only a fixed set of banks can be accessed in each turn. SecMC-NI is designed to allow a security domain to access any bank or rank in its turn while improving the peak bandwidth through interleaving. Memory requests are dynamically scheduled based on each domain's own access pattern.

As in other temporal partitioning schemes, SecMC-NI divides time into turns and uses round-robin scheduling to schedule security domains. Only one security domain can issue requests in each turn. For the convenience of description, we define some parameters as follows:

T_{turn}: Turn length in clock cycles

S: Total number of security domains

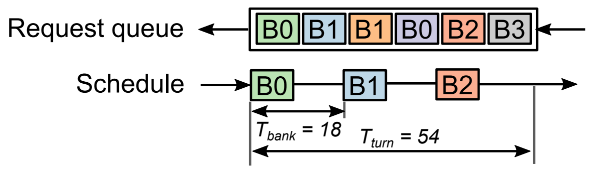


Figure 3: SecMC-NI scheduling example.

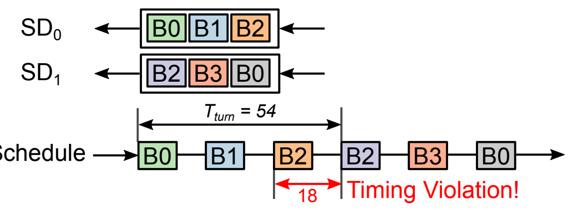


Figure 4: Bank conflict in SecMC-NI scheduling.

T_{bank}: Minimum number of cycles between requests that access different banks

T_{rank}: Minimum number of cycles between requests that access different ranks

Request Selection. In each turn, request selection algorithm picks requests from a security domain to be issued. The requests are chosen based on the following rules. First, requests must access different banks. This rule ensures that we can schedule the chosen requests at a rate of one request every T_{bank} cycles. Second, the algorithm enforces that the maximum number of requests to be scheduled is $\lfloor T_{turn}/T_{bank} \rfloor$. This ensures that the picked requests can fit into one turn. Finally, the selected requests are scheduled at cycle 0, T_{bank}, $2 \cdot T_{bank}$, ..., $(\lfloor T_{turn}/T_{bank} \rfloor - 1) \cdot T_{bank}$ within the turn.

As a concrete example, consider the schedule in Figure 3. The notation Bi indicates a memory request for bank i. Let's assume $T_{turn} = 54$ and $T_{bank} = 18$. Although there are two requests for bank 0 and bank 1, only one of the requests gets issued. Because at most 3 requests can be issued in this turn, the request to bank 3 remains in the queue.

Compared to BTA, SecMC-NI is more flexible. As long as the requests are accessing different banks, they are allowed to be issued together in a turn. However, this additional flexibility also introduces new complexity. Consider the example in Figure 4. Two different security domains are ready to schedule their requests. If the requests are scheduled in their arrival order, the resulting schedule will violate DRAM timing constraints due to a bank conflict. SecMC-NI addresses this problem by reordering memory requests.

Reordering. For reordering, SecMC-NI uses a small buffer to keep track of the schedule in the previous turn. After the requests are selected for the current turn, SecMC-NI checks each of these requests against requests in the previous turn. If Req_0 in the current turn accesses the same bank as Req_1 in the previous turn, Req_0 will be placed at the same relative position as Req_1. As a result, these two requests are separated by T_{turn} cycles. To allow accesses to the same bank in two consecutive turns, T_{turn} must be long enough (43 cycles) to satisfy DRAM timing for accesses to the same bank. After the reordering, the history buffer is updated with the schedule of the current turn. Using this reordering algorithm, SD_1's schedule in Figure 4 becomes $B0, B3, B2$, which satisfies DRAM timing constraints.

Unfortunately, reordering introduces a security concern because it allows one turn's schedule to be affected by the previous turn. To avoid the information leak through reordering and enforce strict non-interference, SecMC-NI delays sending memory responses back to CPU until all memory requests in a turn finish memory accesses. SecMC-NI then sends the memory responses in the arrival order of memory requests.

Interleaving Requests from Different Ranks. So far, we only considered which banks that memory requests access. However, we can construct a far more efficient scheduling if we consider both the rank and bank of a memory request. The basic idea is to construct a separate schedule for each rank, and interleave these schedules to form the final schedule. The request selection algorithm first picks the $\lfloor T_{bank}/T_{rank} \rfloor$ ranks with the most pending requests. For each chosen rank, the algorithm then picks the requests to different banks as described earlier. Once all requests are selected, SecMC-NI interleaves requests from different ranks to construct the final schedule. The ranks needs to be reordered based on the previous turn's schedule to avoid timing violations, similar to the bank reordering described above.

As a concrete example, consider the example in Figure 5 with the following timing parameters: $T_{turn} = 54$, $T_{bank} = 18$, $T_{rank} = 6$. The requests are grouped into separate queues based on which rank they access. SecMC-NI first picks the ranks with the most pending requests. With these timing parameters, at most three ranks can be interleaved. For SD_0, rank {0, 3, 2} are selected to be issued in this turn. For each rank, requests to different banks are selected. Each rank constructs its schedule separately, and these schedules are shifted and combined to construct the final schedule as shown in Figure 5. The notation $RiBj$ indicates a request that accesses bank j in rank i. We separate each schedule by T_{rank} cycles to avoid timing violation between requests to different ranks. For SD_1, rank {2, 1, 3} are selected to be issued. To avoid timing violation, SecMC-NI reorders the rank schedules so that each rank's schedule has no timing violation with the previous turn. This reordering ensures that the requests that access the same bank and rank are separated by at least T_{turn} cycles. In the best case, a security domain can issue 9 requests in a single turn using SecMC-NI, which significantly improves the peak throughput.

Address Randomization. Although SecMC-NI has the same peak bandwidth as rank partitioning, it can only do so if requests are evenly distributed across different ranks and banks. If all requests are accessing the same rank and bank, only one access can be issued per turn. To improve the scheduling efficiency, we add address randomization to SecMC-NI. The randomization maps a physical address to a randomized DRAM address used for scheduling by XORing a random bit vector so that requests are more evenly distributed across ranks and banks.

Security. SecMC-NI completely removes timing channels among security domains because the memory latency of each security domain is independent of accesses from other domains. The scheduling algorithm ensures that the same set of accesses are scheduled for each turn no matter which addresses are accessed by other domains. The ordering within a turn is hidden by delaying responses until all requests in a turn finish.

4. SECMC-BOUND: DYNAMIC SCHEDULING WITH BOUNDED LEAK

4.1 Intuition and Overview

Schemes that completely remove timing interference are inherently inefficient because they need to ensure no interference even for the worst-case traffic. In particular, accesses from different secure domains must be far enough apart (43 cycles) in case they access the same bank, which requires a long turn length. Long turns lead to a significant queueing delay. In practice, however, normal traffic patterns should only experience a small number of bank conflicts.

Here, we introduce a new scheme, named SecMC-Bound, which enables a trade-off between security and performance with an information theoretic bound on leakage. Allowing controlled timing interference enables the scheme to use dynamic scheduling optimized for common-case behaviors. SecMC-Bound defines an expected response-time (ER) for each memory request and delays each response to its ER. The ER is determined only based on requests within a security domain. As a result, timing interferences among security domains are invisible to programs as long as each request return at its ER. An access with significant interference may violate its ER, potentially leaking information. SecMC-Bound limits the number of ER violations by delaying all responses of a security domain to the worst-case response-time (WR), effectively enforcing a completely secure scheme such as TP and SecMC-NI, after a predefined limit on ER violations is reached over each period.

4.2 SecMC-Bound Algorithm

As in other secure memory controller designs, SecMC-Bound assumes that there are per-domain input and output queues. Memory requests are stored in the input queue of the corresponding security domain, and responses from DRAM are stored in the output queue before being returned to the last-level cache (LLC).

Expected Times. Instead of relying on conservative turn-based scheduling to remove interference among memory accesses, SecMC-Bound hides interference by delay-

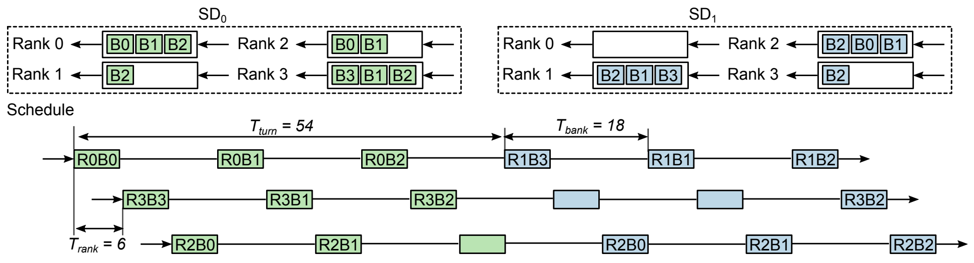


Figure 5: SecMC-NI scheduling with rank interleaving.

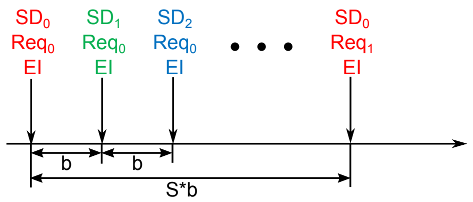


Figure 6: Expected issue times for memory requests.

ing a response to its expected response-time (ER). The high-level approach can be applied in many ways as long as the ER of each memory request is independent of requests from other security domains. Here, we describe one such approach used in this paper.

To assign an ER to a request, we first assign an expected issue-time (EI), which defines the clock cycle when the request is expected to be issued to DRAM. Figure 6 shows an example of expected issue times based on a round-robin schedule with a fixed issue rate; S represents the total number of security domains and b represents the time interval between EIs of two consecutive requests from different security domains. The EI for the i^{th} memory request from security domain s ($t_{EI}(s,i)$) can be calculated by Equation 1.

$$t_{EI}(s,i) = S \cdot b \cdot j + s \cdot b \tag{1}$$

where j is the minimum integer s.t. $t_{EI}(s,i)$ is greater than the time that the request is enqueued ($t_{enq}(s,i)$) and the EI of the previous request ($t_{EI}(s,i-1)$). We calculate ER by simply adding a constant parameter (d) to the corresponding EI as shown in Equation 2.

$$t_{ER}(s,i) = t_{EI}(s,i) + d \tag{2}$$

ER Violation. When a DRAM access is completed, the corresponding response is put into a per-domain output queue and returned to the LLC at its ER. The built-in delay (d) in ER hides small timing interference.

However, there is no guarantee that every request can finish before its ER. If a request cannot be issued early enough to meet its ER, we count it as an ER violation. ER violations represent potential information leakage as they allow an attacker to observe timing variations caused by memory interference. If a request is allowed to be returned at any clock cycle after its ER, an attacker can learn the exact delay value. To limit the amount of information that one ER violation leaks, we restrict the added delay for an ER violation to predetermined values, i.e., d_k ($1 \leq k \leq W-1$). We also specify the worst-case delay, d_W, in a way that a request is

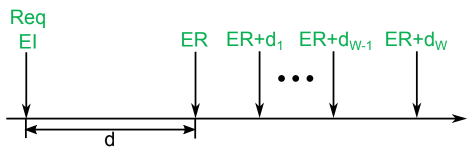


Figure 7: Possible delay values for an access.

guaranteed to finish before $ER + d_W$.

Figure 7 shows the predetermined delay values for a memory request. When a memory access for a request completes, the response is returned at the closest cycle with one of the allowed delay values. Under this scheme, a request that violates ER can only have W possible delays, which limits the amount of information that one ER violation may leak.

When a memory request is issued too late to meet its ER, SecMC-Bound increments a counter (m) to record the number of ER violations. Both EI and ER of the request are also incremented by d_k to account for the visible delay. The following memory request uses this updated EI as the previous request's EI in the constraint ($t_{EI}(s,i) > t_{EI}(s,i-1)$) when calculating its EI.

Worst-Case Times. In order to bound the delay on an ER violation, we need to determine the wort-case response-time (WR) of a request when a memory controller can guarantee to finish the request under any traffic pattern.

The worst-case time can be determined using a secure scheduling algorithm such as TP or SecMC-NI. For example, consider the TP scheme with the minimum turn length, which issues one memory request every T_{turn} cycles. Under the TP scheduling, we can define the worst-case issue-time (WI) for the i^{th} request from security domain s as follows:

$$t_{WI}(s,i) = S \cdot T_{turn} \cdot j + s \cdot T_{turn} + offset_{refresh} \tag{3}$$

where j is the minimum integer such that $t_{WI}(s,i)$ is greater than $t_{enq}(s,i) + T_{turn}$ and $t_{EI}(s,i-1) + S \cdot T_{turn}$. Here, $t_{enq}(s,i)$ is the enqueue time, T_{turn} is the minimum turn length (43) and $offset_{refresh}$ represents the delay due to DRAM refresh cycles. Note that the EI of the previous request ($t_{EI}(s,i-1)$) in the constraint incorporates delays due to ER violations and represents time after the actual issue-time. Therefore, the WI of a request is defined to be at least $S \cdot T_{turn}$ cycles away from the previous request's actual issue-time and at least T_{turn} cycles away from the time that the request is enqueued.

The above construction provides an enough margin for a memory controller to enforce the worst-case time for any traffic patterns. If there is a request whose WI is less than T_{turn} cycles away, our DRAM scheduler enforces a dead time so that no new requests can be issued and all in-flight transactions will be drained. This guarantees that the request can be issued by its WI. Effectively, the scheduling follows TP for one turn. This scheduler design ensures that every request is issued before its WI. In turn, the worst-case response-time (WR) can be computed using a DRAM access latency:

$$t_{WR}(s,i) = t_{WI}(s,i) + t_{RCD} + t_{CAS} + t_{BURST} \quad (4)$$

DRAM Scheduling. SecMC-Bound dynamically schedules memory requests to complete them as soon as possible while prioritizing requests with lower expected issue times and ensuring the worst-case timing. In each clock cycle, the DRAM scheduler selects the request 1) with the lowest EI 2) among ones that can be issued (satisfying all DRAM timing requirements). Note that EIs are only used to prioritize memory accesses in arbitration and the actual issue time of a request does not need to match its EI. For example, a memory request can be issued much earlier than its EI if there no pending requests from other security domains. As an exception, if there is a request whose WI is less than the minimum turn length (T_{turn}), the scheduler enforces the worst-case time as discussed above.

Limiting the Number of ER Violations. In order to bound the information leakage, SecMC-Bound limits the number of ER violations that can happen in a certain time interval. To enforce the limit, SecMC-Bound maintains counters for the number of ER violations (m) and the number of read requests (n) for each security domain, and can be configured to only allow up to M ER violations over a period such as every N read requests or C cycles.

If m reaches the limit M for one security domain, SecMC-Bound switches to the conservative worst-case mode for that security domain and delays every response until its WR. Note that the worst-case delay is applied only to the security domains that reached the limit. As a result, there cannot be any more ER violations for that security domain. Once a period is over, the counters are reset and the output queue again uses ER to delay responses.

4.3 Performance Optimizations

4.3.1 Avoiding Worst-Case Times

When the number of ER violations m reaches the limit M, SecMC-Bound switches to the worst-case mode, which incurs significant performance overhead to that security domain. To reduce the chance of entering the worst-case mode, we can gradually increase the value of d as the number of violations increases. As an example, assume that the limit (M) is 3 violations over 1 million requests. We set the initial value of d to be d_{init} and adjust the value of d based on the number of ER violations (m) for a security domain.

- If $m = 0$, $d = d_{init}$.
- If $m = 1$, $d = d_{init}$ + $delay_1$.
- If $m = 2$, $d = d_{init}$ + $delay_2$.
- If $m = 3$, $d = d_{init}$ + d_W.

Because d increases with m, an ER violation is less likely to happen when m is large. As a result, SecMC-Bound is unlikely to enter the worst-case mode. After a period, the counter m is reset to 0 and d is reset to d_{init}. The optimization can be applied to any value of M by defining d for each possible value of m. Note that a similar optimization can be applied to b to also reduce the chance of entering the worst-case mode.

4.3.2 Dynamic Tuning of d_{init} and b_{init}

SecMC-Bound uses two design parameters d and b to determine expected response times. Intuitively, these two parameters represent different points in the security-performance trade-off space, and we may be able to improve performance with minimal impact on security if we can properly choose d and b depending on application characteristics. For example, applications with infrequent memory accesses may not experience any ER violation even with a small d, which results in a shorter memory latency.

The optimization in Section 4.3.1 adjusts the value of d within each period, but the initial value of d (d_{init}) is still fixed. Here, we discuss how the value of d_{init} can be dynamically adjusted.

In this scheme, we adjust d_{init} at the end of each period based on the number of ER violations (m) observed in that period. The updated d_{init} is used in the following period.

- If $m = 0$, $d_{init} = d_{init}$ - 10.
- If $m \geq M - 1$, $d_{init} = d_{init}$ + 10.
- Otherwise, $d_{init} = d_{init}$.

The above algorithm decreases d_{init} if there was no ER violation in the previous period ($m = 0$), and increases d_{init} if the number of ER violations almost reached the limit ($m \geq M - 1$). The algorithm can be adjusted with different thresholds to change d_{init} more aggressively. For example, a more aggressive scheme may decrease d_{init} when $m < M - 1$. This design can decrease d_{init} to a lower value, but is also likely to result in more ER violations. In our experiments, we used the shown algorithm, which more conservatively change d_{init}.

The same approach can be applied to dynamically adjust the initial value of b (b_{init}). Dynamic tuning of d_{init} and b_{init} has two benefits. First, the dynamic tuning allows the scheme to adapt to different phases of an application. Memory-intensive phases may require large d_{init} and b_{init} to reduce the number of ER violations, while less memory-intensive phases can use smaller d_{init} and b_{init} to improve the performance. Second, the dynamic tuning allows adapting to different workloads without manual designer efforts. The scheme will automatically adjust itself to meet the specified limit on the ER violations for a given workload.

4.4 Information Theoretic Bound

SecMC-Bound is designed so that the response time of each memory request only depends on requests within its security domain, except for ER violations. The added delay on an ER violation is the only property that depends, partially, on memory requests from other security domains, leading to potential timing channels. Here, we present an information theoretic analysis that enables us to conservatively compute a quantitative upper bound on the rate of information leakage based on the number of ER violations.

We start by making the following definitions. Let $\mathbf{x}$ be the history of all memory requests, from all security domains, from time $-\infty$ to the present. Let $\mathbf{Y}$ be the vector of delays $(y_1, y_2, ...y_n)$ seen by a receiver program that places n read requests such that $y_i \in \{0, d_1, ...d_{W-1}, d_W\}$ $\forall i$ s.t. $0 < d_1 < ...d_W$. Note that any $y_i > 0$ is considered an ER violation.

In the context of a covert channel between malicious programs, channel capacity is the most natural information theoretic metric. It can be computed as: $C = \max_x I(\mathbf{X};\mathbf{Y}) = \max_x \{H(\mathbf{Y}) - H(\mathbf{Y}|\mathbf{X})\}$, using the usual information theoretic definition of entropy. Because $\mathbf{Y}$ is deterministic given $\mathbf{X}$ for a memory controller, $I(\mathbf{X};\mathbf{Y}) = H(\mathbf{Y})$. In general, mathematically analyzing the precise distribution of $\mathbf{Y}$ is non-trivial. To simplify the analysis, we conservatively assume that all probability distributions over the possible values of $\mathbf{Y}$ are attainable. In this case, the entropy of $\mathbf{Y}$ is maximized when the distribution is uniform over all $\mathbf{y} \in \mathbf{Y}$ and the channel capacity can be simply computed with the number of possible $\mathbf{y} \in \mathbf{Y}$.

Given that SecMC-Bound limits the ER violations to be at most M out of N requests, the number of possible values of $\mathbf{Y}$ can be computed using a basic combinational analysis. The leakage is simply the logarithm of this value:

$$\max_{\mathbf{X}} I(\mathbf{X};\mathbf{Y}) = \log_2\left(\sum_{m=0}^{M} W^m \binom{N}{m}\right). \tag{5}$$

In the context of an unintentional side channel where a malicious listener snoops on the activity of other programs, maximum leakage, which can be computed as $\mathcal{L}(\mathbf{X} \rightarrow \mathbf{Y}) \equiv \log_2(\sum_y \max_{p(x)>0} p(y|x))$, is a popular metric for the rate of information leakage. In our case where $\mathbf{Y}$ is deterministic in $\mathbf{X}$, maximum leakage also reduces to the logarithm of the number of possible values of $\mathbf{Y}$. Hence, the bound for maximum leakage is the same as the bound for covert channel capacity.

Practical Channel Capacity. The above bound is calculated under conservative assumptions, which are not true in practical systems. The bound assumes that all conceivable $\mathbf{Y}$ distributions are possible and that a malicious program can choose any of these distributions at will. However, in practice, memory accesses are much more likely to result in no ER violation or a low delay even on an ER violation. The bound also captures the information leak from *all* programs to a receiver even though attackers in practice only control a subset of programs and other accesses add uncontrolled noise. The bound also assumes that an attacker can control and measure all memory accesses at a cycle granularity. This is unlikely in practice. For example, caches affect memory requests and it is difficult to maintain perfect synchronization between concurrent programs.

In that sense, we provide a conservative bound for channel capacity. The practically achievable channel capacity is likely be at least two to three orders of magnitude lower than our information theoretic bound under ideal assumptions. For example, a previous study [3] reported the covert-channel capacity around 500 Kbps based on memory contention even when every memory access could be used to leak information in theory. Fortunately, our experimental results show that even our conservative bound can be used to provide good performance with a guarantee on low information leakage. We leave the further refining the bound considering practical limitations as future work.

Processor	
ISA	x86
Core count and frequency	8-core, 2.0GHz
ROB size	128
Issue width	4
Cache	
L1 I-cache	32KB/4-way
L1 D-cache	32KB/4-way
L2 Cache	8MB/64-way, way partitioned
DRAM	
DRAM bus frequency	667MHz
DRAM configuration	1 channel, 8 ranks, 8 banks/rank
Total capacity	16GB
DRAM Timing Parameters (DRAM cycles) $t_{RC} = 34, t_{RCD} = 10, t_{RAS} = 24, t_{FAW} = 20, t_{WR} = 10,$ $t_{RP} = 10, t_{RTRS} = 1, t_{CAS} = 10, t_{RTP} = 5, t_{BURST} = 4,$ $t_{CCD} = 4, t_{WTR} = 5, t_{RRD} = 4, t_{REFI} = 7.8us, t_{RFC} = 107$	

Table 1: Configuration parameters for ZSim and DRAMSim2 simulators.

5. EVALUATION

5.1 Methodology

We used ZSim [13] integrated with DRAMSim2 [14] to evaluate the performance of the proposed schemes. We model a multi-core processor with 8 cores. Each core has a private L1 cache. An L2 cache is shared but is partitioned to remove interference among cores. This configuration was chosen to evaluate the impact of memory timing channel protection schemes without the impact of cache interference. We model a single memory channel with 8 ranks and 8 banks per rank. The detailed configuration parameters are shown in Table 1.

We use multi-program workloads constructed from SPEC CPU2006 benchmark suites to evaluate the performance. To evaluate the performance depending on memory intensity, we use eight copies of the same program on the 8-core processor. For the performance evaluation running multiple copies of one benchmark, we used 24 SPEC benchmarks but show only 12 representative benchmarks in graphs due to the space limit. The benchmarks are ordered based on the memory intensity (misses per kilo instructions) with more memory intensive benchmarks on the right. The geometric mean on each figure is calculated using all 24 benchmarks.

mix-1	ton les gam gro gam h264 mcf bwa
mix-2	gcc cac h264 zeu gam cac gam ton
mix-3	sje per zeu gam les les gcc gcc
mix-4	xal gro gcc bzi dea gob ton hmm
mix-5	nam zeu gam per omn gcc ton zeu
mix-6	per xal cac dea gob les cac gob
mix-7	per gcc cac mcf zeu per omn lib
mix-8	sop gam mil gro hmm sje les lib
mix-9	ton xal omn gcc ton h264 h264 hmm
mix-10	zeu hmm lbm les mcf mcf mcf bwa

Table 2: Mixed workloads.

To understand the performance for more diverse workloads, we use mixed workloads, each of which consists of 8 randomly selected SPEC benchmarks. The 10 mixed workloads that we used are shown in Table 2.

The experiments assume that each core represents a different security domain. Each program fast-forwards for 1 billion instructions and then enters detailed simulation mode in which the memory requests are simulated in cycle-accurate manner. The simulation terminates when all programs have executed 100 million instructions. We only take the first 100 million instructions of each program to calculate the IPC.

We use weighted speedup as the performance metric. For a system with concurrently running programs, weighted speedup is defined as the sum of each program's IPC normalized to the IPC when the program is running by itself:

$$Weighted\ Speedup = \Sigma(IPC_i/SingleIPC_i) \qquad (6)$$

The weighted speedups are normalized to the insecure baseline, which is FR-FCFS [15] in our experiments.

5.2 Performance of SecMC-NI

5.2.1 *Effect of Turn Length*

We evaluate the performance of SecMC-NI with two different turn lengths: 43 cycles and 54 cycles. The 43-cycle turn allows up to 6 requests per turn (3 ranks, 2 banks per rank). The 54-cycle turn allows up to 9 requests per turn (3 ranks, 3 banks per rank). Figure 8 shows the performance comparison between the two turn lengths. For programs with low memory intensity, BTA and SecMC-NI perform almost equally well, reaching 80% of the baseline's performance. However, for memory-intensive programs, SecMC-NI significantly outperforms BTA, almost doubling the performance of BTA in many cases. This is because SecMC-NI allows more flexible scheduling with a higher peak throughput by interleaving ranks and banks.

To better understand the results, Figure 9 shows the average queueing delay of memory requests under each scheme. The queueing delay is calculated as the difference between the time that a memory request arrives at a request queue and the time that the memory request gets issued to DRAM. Due to the inflexible schedule, BTA incurs high queueing delays (~600 cycles). SecMC-NI-43 cuts this queueing delay by half.

Comparison between SecMC-NI-43 and SecMC-NI-54 leads to another interesting observation. Although SecMC-NI-54 has a higher theoretical bandwidth utilization than SecMC-NI-43, the longer turn length of SecMC-NI-54 increases the average queueing delay of

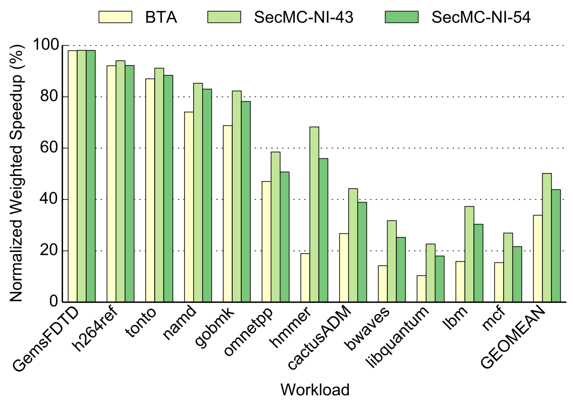


Figure 8: Performance comparison between SecMC and BTA.

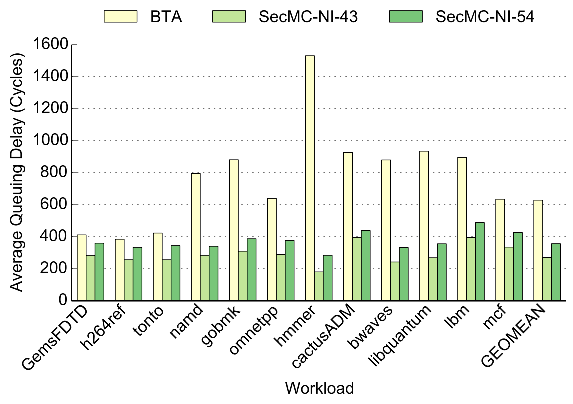


Figure 9: Queuing delay comparison between SecMC and BTA.

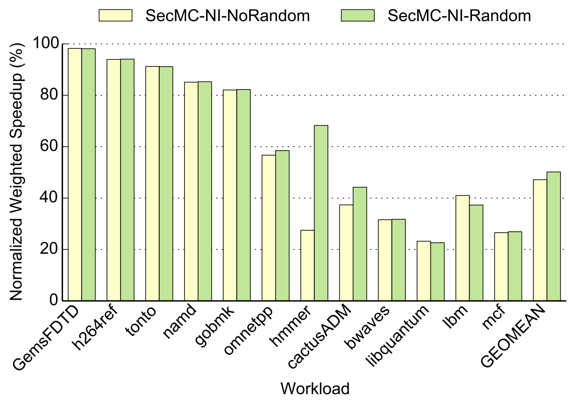


Figure 10: SecMC performance with and without address randomization.

memory requests. Hence, we see SecMC-NI-43 actually outperforms SecMC-NI-54. For the rest of the evaluation section, we use 43 cycles as the turn length for SecMC-NI.

5.2.2 *Effect of Address Randomization*

We then study the impact of address randomization on the performance of SecMC-NI. The purpose of address randomization is to distribute memory requests more evenly across different ranks and banks, thus al-

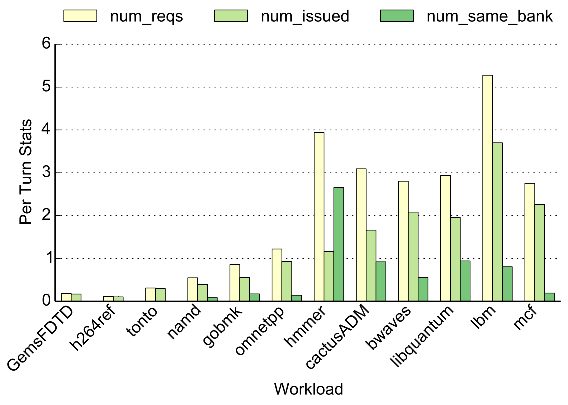


Figure 11: SecMem scheduling statistics.

lowing SecMC-NI to issue more requests in a turn. Figure 10 compares the performance of SecMC-NI with and without address randomization. On average, adding address randomization only improves the performance of SecMC by 2%. However, for specific benchmarks such as *hmmer*, the improvement is significant.

To understand why *hmmer* benefits significantly from address randomization, we profiled the simulations to record scheduling statistics in each turn. Figure 11 shows three scheduling statistics. *num_reqs* represents the average number of requests in the queue for the active security domain in each turn. *num_issued* represents the average number of requests that are issued in each turn. *num_same_bank* represents the average number of requests that try to access the same bank in the same rank. The value of *num_same_bank* indicates the number of requests that cannot be scheduled together in a turn. As the figure shows, *num_same_bank* is large for *hmmer*, meaning that a lot of the requests in *hmmer* have bank conflict. With the help of address randomization, *hmmer*'s requests are distributed more evenly across different banks, which explains the large performance gain for *hmmer* when address randomization is enabled. For programs with many bank conflicts, address randomization is an effective way to improve the performance.

5.3 Performance of SecMC-Bound

5.3.1 Parameter Sweep for b and d

To explore the design space, we tried different values for the design parameters, b and d. We used {3, 6, 18} for b and {64, 128, 256} for d. Note that in these experiments, we do not restrict the number of violations over a period. The experiments also *do not* use optimizations to dynamically adjust b or d. We configure W to be 3 with $d_1 = 30$, $d_2 = 160$ and d_3 to be the worst-case delay. The performance was normalized to the insecure baseline.

Figure 12 shows the performance of SecMC-Bound across different parameter values. We use the notation SecMC-Bound-b-d to represent the scheme with different b and d. For a fixed b, the performance of SecMC-Bound decreases as d increases because a smaller d leads to an earlier expected response-time (ER). A large d increases the memory latency. For a fixed d, the performance decreases as b increases. This is because b affects the expected issue-time (EI), hence also indirectly affecting the expected response-time (ER). A larger b leads to a later expected response-time.

In summary, small b and d improve the performance of SecMC-Bound. As an example, SecMC-Bound-3-64 shows nearly 90% of the baseline's performance, which even beats rank partitioning. However, the high performance of SecMC-Bound-3-64 comes with a lower security level. To see how the security is affected, Figure 13 shows the average number of ER violations per memory request. Note that the y-axis uses a log scale. As the figure shows, small b and d leads to more frequent ER violations, which indicates higher potential information leak. Fortunately, the number of ER violations decreases exponentially as we increase d. Even though a large d leads to lower performance, SecMC-Bound-3-256 still achieves about 60% of the baseline's performance, which is better than SecMC-NI and BTA.

Figure 12 and 13 show that SecMC-Bound provides a large trade-off space between performance and security. Users can tune the performance of their memory controller based on how much security they are willing to sacrifice.

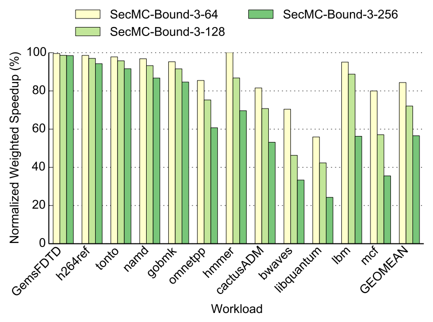
5.3.2 Limiting the ER Violations

We study the performance of SecMC-Bound when we apply the mechanism to limit the number of violations in a period. Figure 14 shows the performance results using $b = 6$ and $d = 160$. We use four different limits. "4 in 1,000" means we allow 4 ER violations in every 1,000 requests. Once the number of violations for a security domain reaches the limit within a period, this security domain enters the worst-case mode in which it always delays responses until their worst-case response-time (WR). The figure shows that enforcing a lower limit on the number of ER violations increases the performance overhead. As the limit gets lower, a security domain is more likely to enter the worst-case mode, which lowers its performance.

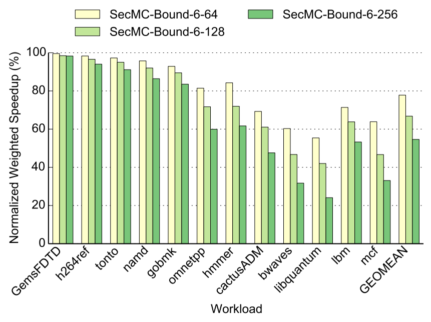
Yet, enforcing a limit on the number of ER violations is necessary to provide a bound on the information leak. Figure 15 shows the leakage bound for each limit. As we lower the limit, the leakage bound drops accordingly. A designer may choose the limit based on the assets he or she wants to protect. For example, if the asset is a large file such as an HD movie, even leaking hundreds of bits per second may be acceptable. On the other hand, if the scheme needs to protect a small secret such as a cryptographic key, the limit will need to be much lower.

5.3.3 Optimization 1: Avoiding Worst-Case Times

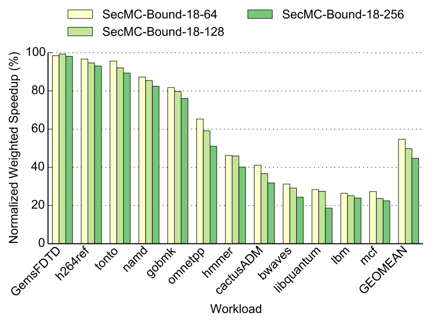
The high performance overhead for the low limit in Figure 14 is mainly caused by the worst-case mode where all responses are delayed to their WR. One of the proposed optimizations reduces the chance of reaching the ER violation limit and entering the worst-case mode by gradually increasing the value of d. To see the effectiveness of this optimization, we ran experiments with this optimization implemented—the value of d increases by



(a) b=3

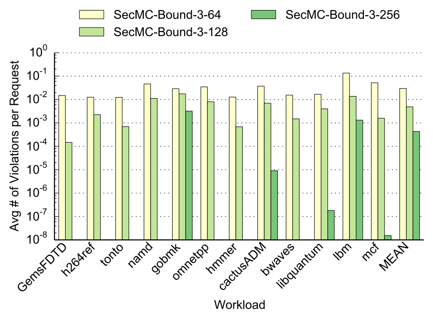


(b) b=6

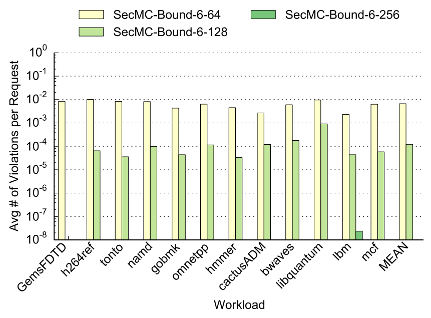


(c) b=18

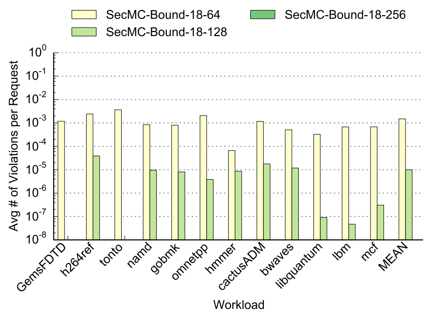
Figure 12: SecMC-Bound performance for a range of b and d values.



(a) b=3



(b) b=6



(c) b=18

Figure 13: The number of ER violations for a range of b and d values.

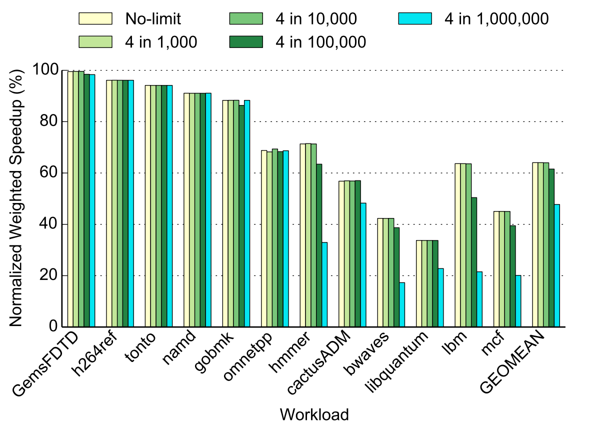


Figure 14: Performance with a limit on ER violations.

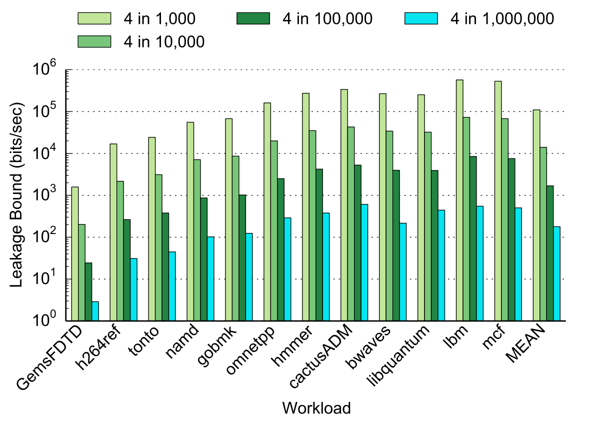


Figure 15: Leakage bound with a limit on ER violations.

10 whenever an ER violation happens within each period. Figure 16 shows the performance results with this optimization. Compared to the results without the optimization shown in Figure 14, the performance drastically increases for the low ER violation limit (4 in 1,000,000). The result suggests that this optimization is effective in preventing a security domain from frequently entering the worst-case mode when the limit is low. However, the performance is slightly degraded for cases with a high limit. Security domains do not often reach the ER violation limit in these cases, hence increasing the value of d actually increases the average memory latency. The results suggest that the optimization should be used when a security domain is likely to reach the limit on ER violations in a period.

5.3.4 *Optimization 2: Dynamic Tuning of d_{init}*

We study the performance impact of dynamically tuning the d_{init} value based on the number of ER violations in the previous period. In this experiment, the initial value of d (d_{init}) is set to 160 and b is set to 6. Figure 17 shows the performance with this optimization under different ER violation limits. When the limit is high (4 in 1,000), we see a noticeable performance improvement over the static case with a fixed d_{init}. This is because the optimization drastically reduces the value of d_{init} (e.g., from 160 to 72) while the static case uses a fixed d_{init} value (160). However, when the limit becomes lower, the performance improvement gradually drops down to zero. For a lower limit with a longer period, m over a period is more likely to be non-zero and d_{init} often cannot be reduced.

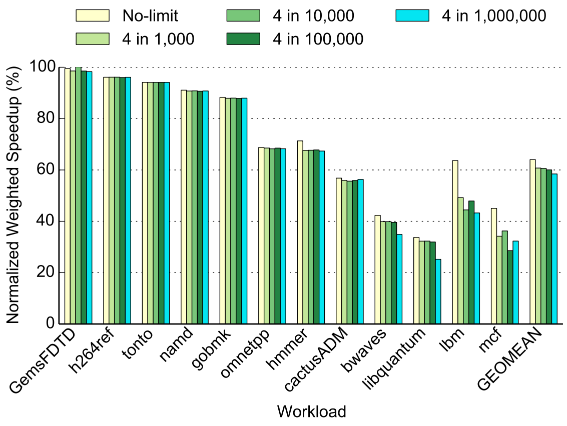


Figure 16: Performance with the optimization to avoid the worst-case mode.

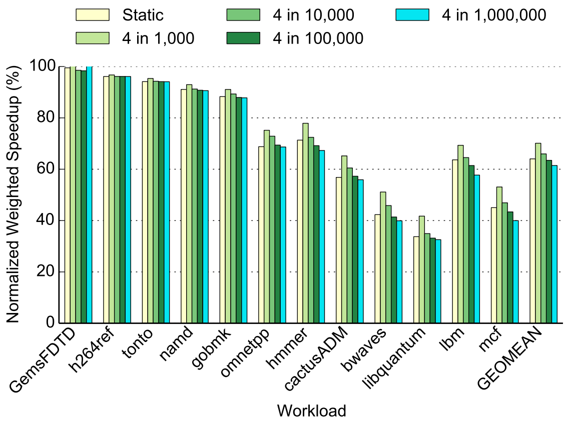


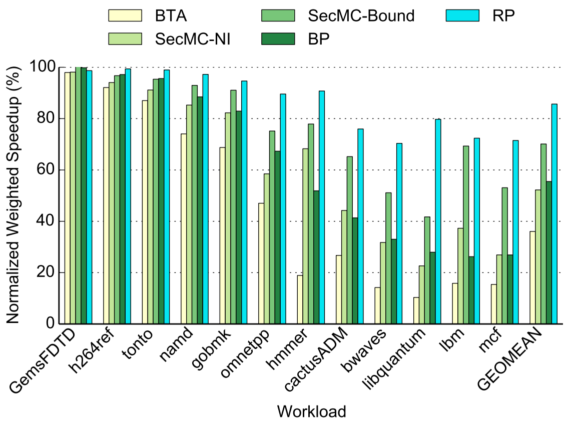
Figure 17: Performance with the d_{init} tuning.

5.4 Overall Performance Comparison

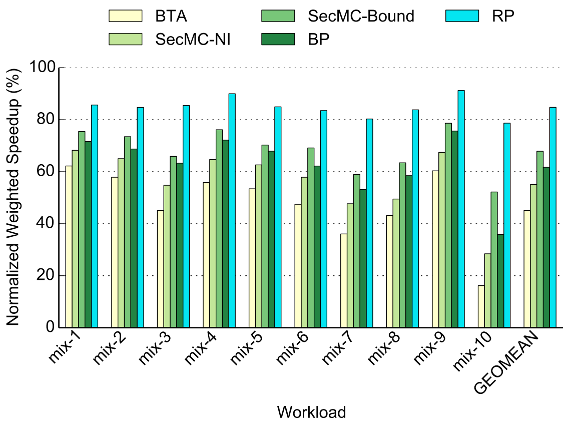
Figure 18a shows the overall performance comparison among SecMC-NI, SecMC-Bound, and previously proposed secure memory scheduling schemes. In this experiment, we use the case with $b = 6$ and $d = 160$ to represent the performance of SecMC-Bound. The performance of SecMC-NI is significantly higher than BTA and close to BP. SecMC-Bound achieves 70% of the insecure baseline performance, outperforming BTA, SecMC-NI and even BP. The performance benefit shows that SecMC-Bound allows more flexible memory scheduling than completely secure schemes while providing a theoretic bound on information leakage.

To understand the performance of the SecMC schemes on more realistic workloads, we repeated all experiments using mixed workloads which consist of different SPEC benchmarks. Due to the space limit, we only show the overall performance results for the mixed workloads in Figure 18b. The figure shows the same overall performance trends with the one shown in Figure 18a; SecMC-NI outperforms BTA and is close to BP while SecMC-Bound outperforms both BTA and BP. The other experimental results on the mixed workloads also show the same high-level trends with the single-benchmark workload results shown in this paper.

5.5 Summary



(a) Single-benchmark workload.



(b) Mixed workload.

Figure 18: The overall performance comparison.

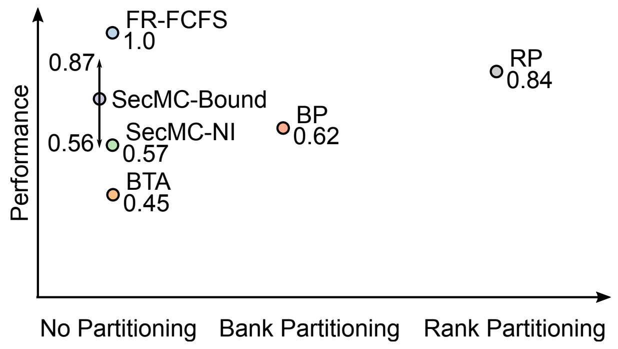


Figure 19: Design space summary.

Figure 19 shows the summary of the secure memory controller design space using the performance for the mixed workloads. SecMC schemes do not require spatial partitioning and outperform BTA, which represent the best previous scheme without spatial partitioning. SecMC-NI achieves similar performance as bank partitioning. SecMC-Bound's performance spans from 56% to 87% of the baseline, depending on the values chosen for b and d, enabling a trade-off between performance and security with a bounded information leakage. While not shown here, we found that SecMC-

Bound can also be combined with spatial partitioning to further improve their performance.

6. RELATED WORK

Previous studies have demonstrated timing-channel attacks and proposed protection techniques for various hardware resources, including caches [16, 17, 18, 7, 8, 9], on-chip networks [10, 11], and memory controllers [2, 12, 4]. Most of these protection techniques rely on temporal partitioning or spatial partitioning to eliminate the interference between different security domains. Researchers also proposed solutions to mitigate timing channel attacks by injecting noise [19] or restricting sensitive operations [20]. In this paper, we propose new protection techniques for shared memory controllers that reduces overhead of existing approaches while still providing strong security guarantees.

Zhang et al. [21] proposed predictive mitigation technique, which exponentially increases observable timing to bound information leakage through program execution time. Fletcher et al. [22] applied a similar technique to ORAM [23]. While we borrow the idea to delay observable timing from the previous work, this paper proposes a new scheme for shared memory controllers.

7. CONCLUSION

This paper shows that the overhead of timing-channel protection can be significantly reduced while providing strong security guarantees. SecMC-NI shows that the scheduling for temporal partitioning can be made far more efficient using dynamic scheduling and re-ordering while still removing timing channels. SecMC-Bound shows that trading off security for performance can further improve performance. In particular, the paper shows that a quantitative bound can be provided for the information leakage under this scheme.

8. 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.

9. REFERENCES

[1] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[2] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[3] C. Hunger, M. Kazdagli, A. S. Rawat, A. G. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding Contention-Based Channels and Using Them for Defense," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[4] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari, "Avoiding information leakage in the memory controller with fixed service policies," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[5] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," 2005.

[6] J. Kong, O. Aciicmez, J. P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[7] 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.

[8] 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.

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

[10] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, 2012.

[11] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[12] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh, "Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller," in *2016 IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[13] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[14] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, 2011.

[15] W. Zuravleff and T. Robinson, "Controller for a synchronous dram that maximizes throughput by allowing memory requests and commands to be issued out of order," May 13 1997. US Patent 5,630,096.

[16] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[17] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005.

[18] D. J. Bernstein, "Cache-timing attacks on AES," tech. rep., 2005.

[19] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[20] B. Saltaformaggio, D. Xu, and X. Zhang, "BusMonitor: A hypervisor-based solution for memory bus covert channels," in *Proceedings of 6th European Workshop on Systems Security (EuroSec)*, 2013.

[21] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[22] C. W. Fletchery, L. Ren, X. Yu, M. V. Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[23] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43.