Timing Channel Protection for a Shared Memory Controller

Yao Wang, Andrew Ferraiuolo, and G. Edward Suh*
Cornell University
Ithaca, NY 14850, USA
{yw438,af433,gs272}@cornell.edu

Abstract

This paper proposes a new memory controller design that enables secure sharing of main memory among mutually mistrusting parties by eliminating memory timing channels. This study demonstrates that shared memory controllers are vulnerable to both side channel and covert channel attacks that exploit memory interference as timing channels. To address this vulnerability, we identify the sources of interference in a conventional memory controller design, and propose a protection scheme to eliminate the interference across security domains through two main changes: (i) a per security domain based queueing structure, and (ii) static allocation of time slots in the scheduling algorithm. Multi-programmed workloads comprised of SPEC2006 benchmarks were used to evaluate the protection scheme. The results show that the proposed scheme completely eliminates the timing channels in the shared memory with small hardware and performance overheads.

1. Introduction

Modern computing systems are becoming increasingly vulnerable to timing channel attacks that leak information through interference in shared resources. For example, in cloud computing, clients often need to share hardware resources with untrusted parties - potentially their competitors or malicious users - in order to benefit from the flexibility and cost efficiency of having a large pool of physical resources. Unfortunately, shared resources introduce timing channels among virtual machines (VMs) that can be used to extract secrets from other VMs or create unauthorized communication channels between colluding VMs. Similarly, downloaded applications that cannot be fully trusted may perform side-channel attacks while running on the same device as trusted applications with confidential information.

While timing channel attacks and their countermeasures have been studied in the context of shared caches [15, 4, 16, 14, 26, 27] and on-chip networks [24, 28], to the best of our knowledge, timing channels through a shared memory channel have not been studied at the hardware architecture level. Like cache timing channels, a memory-based timing channel attack can be carried out without physical access to the hardware, because the memory latencies of one program depend on memory accesses from other programs sharing the memory.

In this paper, we demonstrate that memory timing channels exist for multi-core systems, and propose an efficient protection scheme to completely eliminate them. In a shared memory controller, the time that one memory request is scheduled depends on other competing requests. Thus, there exists a memory timing channel between software modules in multiple security domains. This timing channel can be exploited by an adversary to carry out either a side-channel attack (where a malicious software module measures its own memory timing to learn a secret used by a program in another security domain), or a covert-channel attack (where colluding programs in two different security domains leak information to each another despite restrictions on explicit communication).

In order to develop a protection scheme, we first study sources of interference in a memory controller, and categorize them into three groups: queueing structure interference, scheduler arbitration interference, and DRAM device resource contention. Broadly, interference is caused by multiple programs that access the memory concurrently, allowing memory requests from different programs to affect the timing of others. The goal of the protection scheme is to eliminate memory interference among security domains, which contain one or more software modules such as processes in traditional systems and virtual machines in cloud computing.

We present an approach to prevent memory interference which we refer to as temporal partitioning (TP). Temporal partitioning groups requests in queues according to the security domain they belong to. Then, a fixed time period, called a turn, is statically allocated to each domain in a time shared fashion so that a memory controller only schedules requests from one security domain. At the end of each turn, a short window of time during which no memory transaction can issue is added to prevent two timing channels: one caused by interference between the previous and current turns, and another caused by refresh operations delayed by memory requests. Since only requests from one active domain can be scheduled during each turn, there cannot be any cross-domain interference and all memory timing channels are eliminated.

Experimental results suggest that the execution time overhead for temporal partitioning is only 1.5% on average using

*The first two authors contributed equally to the work.

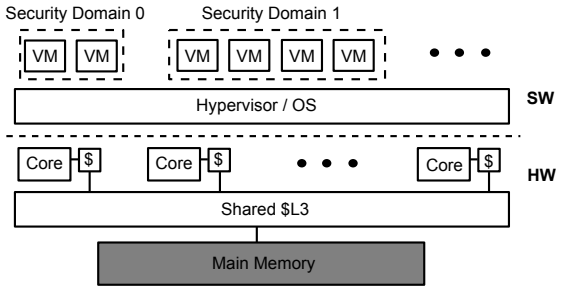

Figure 1. Problem setup: cloud computing example.

in-order core, and 1.4% using out-of-order core when two security domains share a memory controller running SPEC 2006 benchmarks. Temporal partitioning only requires simple changes to the memory controller with a small mount of additional hardware resources: a revised queueing structure, a counter, and a small amount of combinational logic to restrict scheduling decisions.

The rest of the paper is organized as follows. Section 2 discusses the memory timing channel problem. Section 3 analyzes a baseline memory controller for timing channel violations and presents the temporal partitioning scheme. Section 4 evaluates the security properties and execution time overheads of temporal partitioning experimentally. Section 5 discusses related work. The paper concludes in Section 6.

2. Timing Channels in Shared Memory

2.1. Problem Setup

Figure 1 shows the problem setup using cloud computing as an example. We consider a multi-core platform where multiple processing cores share one or more memory controller(s) and the attached off-chip memory (DRAM). The platform also includes a cache hierarchy that consists of private and shared caches. The hardware is managed by a privileged software layer such as a hypervisor and an OS, and is shared by multiple software modules, such as virtual machines and user applications, which run in parallel.

In this work, we assume that the management software such as a hypervisor is trustworthy and properly controls explicit communication channels. Also, we assume that the platform cannot be physically accessed by an adversary. However, an adversary is capable of running an arbitrary program in a way that the attack program shares the memory controller with a target victim program. For example, in cloud computing, a virtual machine of one client can co-reside with virtual machines of others. A recent study [16] has demonstrated an attack on EC2 that allows a malicious virtual machine forcing to be co-located with a target virtual machine. An attacker can also exploit client parameters to infer a reduced search space of physical machine locations.

The goal of the protection mechanism is to eliminate timing channels between security domains through a shared memory controller. A security domain can include one or more software modules such as VMs, processes, and threads that can share the same timing-channel protection. In the cloud computing scenario, a security domain may consist of VMs that belong to the same user. Different security domains can be owned by mutually distrusting users who wish to keep secrets from one another despite using shared resources.

2.2. Memory Timing Channel Attacks

This work considers two broad classes of attacks that exploit memory timing channels. The goal of a side-channel attack is to gain access to secret information possessed by the victim, which does not intend to leak the secret. The adversary can intentionally create contention in the memory controller and make performance measurements on its own operations to learn about memory accesses from the victim, which the attacker hopes will correlate to a secret.

In a covert-channel attack, the adversary already possesses a secret, but is limited in how it can share this secret. For example, a malicious 3rd party web application may try to leak a user's data when the cloud infrastructure such as Amazon EC2 restricts its network connections so that it can directly communicate only with the user. The adversary can try to bypass such restrictions using a timing channel to another co-residing VM whose network connection is not restricted. For example, the adversary may collude with a co-residing VM, and communicate the secret by deliberately modifying its workload to cause a timing variation in the colluding VM's memory accesses.

We note that a protection scheme needs to remove interference between security domains for complete protection against both side-channel and covert-channel attacks. Obfuscation techniques such as randomization or noise injection are insufficient to prevent intentional information leaks in covert-channel attacks since random noise can be removed statistically.

2.3. Example Attacks

2.3.1. Side-Channel Attack on RSA. As an example, this side channel attack shows how a private key of an RSA decryption program can be compromised by exploiting the interference in memory accesses. The system setup is shown in Figure 2. The system has two cores, each with a private direct-mapped L1 cache. The RSA decryption algorithm runs on Core 0 while an attack program is running simultaneously on Core 1.

The RSA decryption algorithm, uses a private key to decrypt an encrypted message. It is often implemented with the square and multiply algorithm to perform fast exponentiation. In this implementation, the bits in the private key are checked one by one, and a modulo operation is performed only when the bit is "1". In this attack example, the memory addresses are configured so that when this modulo operation is performed, a cache miss occurs. In other words, the

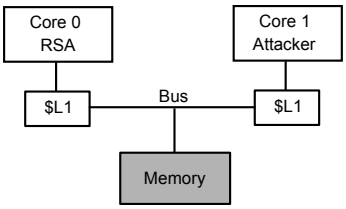

Figure 2. System setup for the RSA attack.

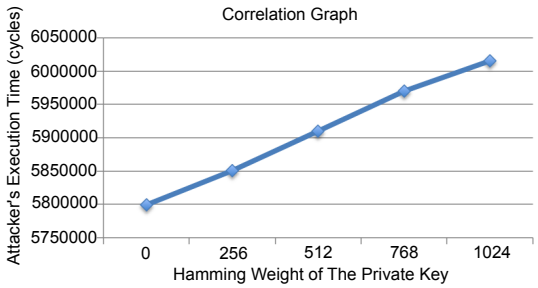

Figure 3. RSA side-channel example.

number of memory requests for the RSA algorithm is directly dependent on the number of "1" bits, or the Hamming weight, of the private key. The attacker issues memory requests to the DRAM continuously and measures the time to finish those requests.

Figure 3 shows the execution time of the attack program as a function of the Hamming weight of the private key. As can be seen, the attacker's execution time has a direct correlation with the Hamming weight, meaning that the attacker can estimate the number of 1s in the private key by simply measuring its own execution time.

2.3.2. Covert-Channel Attack. In this shared memory covert-channel example, one adversary tries to send information to another adversary despite measures to prevent this communication. The system setup is similar to Figure 2 except that now Adversary 0 runs on Core 0 and Adversary 1 runs on Core 1. The goal of Adversary 0 is to send the sequence "10010110" to Adversary 1. Adversary 0 achieves this goal by dynamically changing the memory demand, which affects the latency of memory requests from Adversary 1. To send a "0", Adversary 0 does not issue any memory requests for a period of time. To send a "1", Adversary 0 sends many memory requests. Meanwhile, Adversary 1 keeps sending memory requests and tracks the dynamic throughput it can achieve using a software counter.

Figure 4 shows the memory throughput observed by Adversary 1 over the last 5,000 cycles. As can be seen, the throughput shows a pattern that corresponds to the bit stream that Adversary 0 intends to send. When the throughput is low, Adversary 1 can infer that Adversary 0 is sending a lot of memory requests, and interprets the bit being sent as a "1". Otherwise, the bit being sent is a "0". Using the interference in the memory, Adversary 1 can fully recover

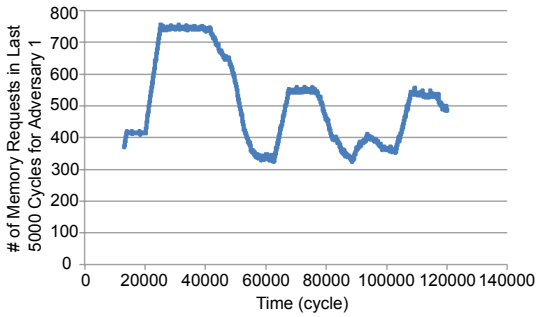

Figure 4. Covert-channel example.

the information that Adversary 0 sends, proving the success of this covert channel attack.

3. Protection Scheme

3.1. Objective

Memory requests from different security domains contend for the same shared resources and can affect the latency of each other, which opens a timing channel. The objective of our protection scheme is to eliminate interference among memory accesses from different security domains. In other words, the timing of each memory access from one security domain should always be independent of memory accesses from other security domains. This property guarantees that the memory timing of one security domain cannot be measured to learn anything about another security domain.

3.2. Baseline Memory Controller

Before discussing our protection scheme, we first describe the conventional memory controller architecture and identify the sources of interference in the design. Figure 5 shows the architecture of a conventional memory controller. One memory access takes the following steps: (i) it is enqueued into one of the request queues based on the address, (ii) it wins bank arbitration, (iii) it wins transaction scheduler arbitration, and (iv) it gets sent to the DRAM device. The First-Ready First-Come First Served (FR-FCFS) [17] scheduling algorithm is used for the baseline memory controller. As shown below, there are three sources of interference in the baseline memory controller.

3.2.1. Queueing Structure Interference. The baseline memory controller has a separate queue for each combination of the ranks and banks (e.g. if there are 3 ranks and 4 banks, a typical memory controller will have 12 queues). This ensures that requests for each bank are put into a separate queue. Although this queueing structure is beneficial for exploiting bank-level parallelism in DRAM accesses, it introduces interference among memory accesses from different security domains. In this queueing structure, a queue can mix memory requests from different security domains, which are denoted by different patterns in Figure 5. As shown in Figure 6a, Request A from security domain 0 can be delayed in the queue by Request B from another security

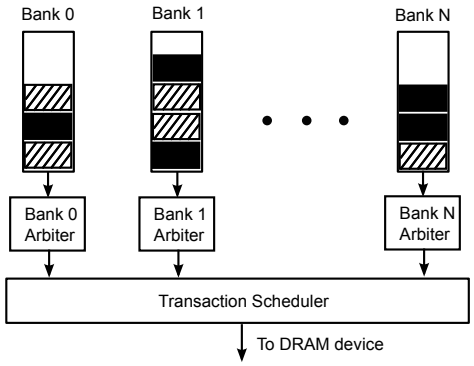

Figure 5. Conventional memory controller.

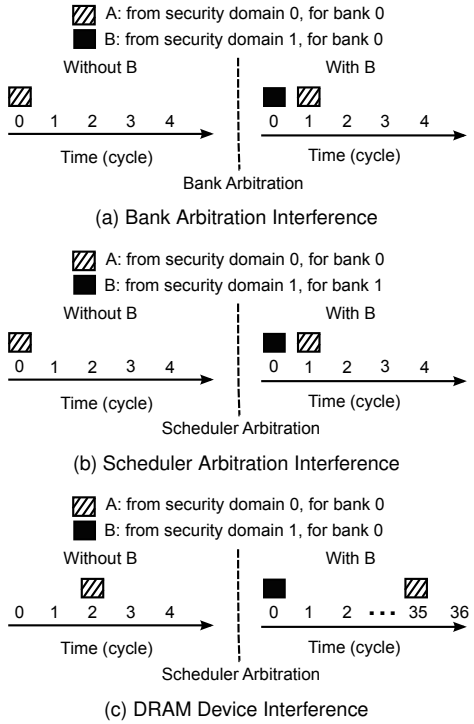

Figure 6. Interference in memory controllers.

domain 1 if the bank arbiter schedules Request B prior to Request A.

Interference in the queueing structure also occurs whenever the memory controller stalls the requests to a particular bank when that bank's request queue is full. If security domain 0 fills the request queue of bank 0 and stalls the memory requests from security domain 1, security domain 1 can learn that security domain 0 is sending many memory requests to bank 0.

3.2.2. Scheduler Arbitration Interference. The transaction scheduler also causes interference. As can be seen in Figure 6b, suppose Request A and Request B are for different banks and they both win bank arbitration in cycle 0. Without Request B, Request A wins the scheduler arbitration and is sent to the DRAM at cycle 0. However, if Request B exists and arrives in the queue earlier than Request A, the FR-FCFS scheduler will favor Request B in arbitration, thus delaying Request A to the next cycle. This changes the timing of Request A.

3.2.3. DRAM Device Interference. Resource contention in DRAM device components such as the command bus, the data bus, banks, and ranks can also cause timing channels. For example, assume Request A and B are from different security domains and intend to access the same bank. Request A arrives at the queue at cycle 2 and Request B arrives at cycle 0. Without Request B, Request A wins bank arbitration and scheduler arbitration at cycle 2. However, if Request B exists and is scheduled at cycle 0, Request A cannot win scheduler arbitration at cycle 2 even if it wins bank arbitration, because the DRAM device cannot serve two memory requests to the same bank concurrently. In an open page policy, if the second request is a row hit, it needs to wait until the first request finishes I/O gating. In a close page policy, the second request needs to wait even longer, because it cannot be scheduled until the bitline is precharged. As shown in Figure 6c, Request A is not scheduled until cycle 35 because the bank has been busy.

The interference problem is not limited to the FR-FCFS scheduling algorithm, but exists for most memory scheduling algorithms (i) because queueing structures mix requests from different security domains, (ii) because the arbitration of the transaction scheduler depends on the dynamic demands of different security domains, (iii) and because of the properties of the DRAM device. All these sources of interference can be used as timing channels to derive the memory usage characteristics of security domains, and leak secret information. With the sources of interference identified, we now describe the protection scheme to eliminate these timing channels.

3.3. Protection Mechanisms

3.3.1. Queueing Structure Protection. To prevent interference among memory requests from different security domains in the same request queue, the queueing structure proposed in this work includes queues for each combination of ranks and security domains and not for each combination of ranks and banks. Figure 7 shows the new queueing structure. With per-security domain queueing, memory requests from different security domains are separated and stored in different queues, and therefore, bank arbitration cannot cause interference among them. Interference can still exist between requests in the same queue, however, they belong to the same security domain, so the interference is benign. In order to exploit bank parallelism, this scheme also requires scheduling logic that scans the queue for requests to an idle bank. Similar logic is also used in a conventional open page memory controller to find requests to open rows.

3.3.2. Scheduling Protection. Concurrent memory accesses from multiple security domains cause both arbitration inter-

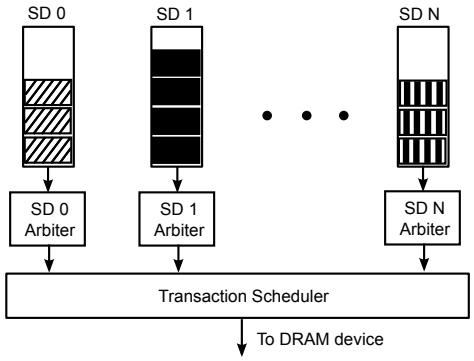

Figure 7. Queueing structure per security domain.

Figure 8. Static time-slot allocation in temporal partitioning.

ference and DRAM device interference. These two types of interference can be eliminated if only one security domain uses memory resources at a time. Thus, we propose Temporal Partitioning (TP) that divides the time into fixed-length turns during which only requests from a particular security domain, which we say is active, can issue.

Figure 8 illustrates the high-level approach of TP. The length of a turn is defined as T_{turn}. During each turn, requests of the active security domain are scheduled normally, but requests from other domains are not allowed. While requests within each domain can cause interference with each other, such intra-domain interference is benign as they cannot leak information to another domain. TP allows the full memory bandwidth utilization of the baseline memory controller provided the active security domain has a sufficient number of requests to take advantage of its turn (i.e. bandwidth is wasted when there is no memory request that can be issued from the active security domain but there are requests in an inactive domain that could normally be issued). At the end of each turn, the next security domain is selected using a fixed, static schedule and activated. The implementation discussed in this work uses a round-robin static schedule, however, any other static schedule will suffice.

3.3.3. Row Buffer Policy. In a DRAM cell, requests to data already in the row buffer (sense amplifier) are much faster than others. In an open page row-buffer management policy, the most recently activated row is left in the row buffer of the bank until another row in that bank must be accessed. This is beneficial for workloads that have a lot of row locality (and are therefore likely to reuse data already in the row buffer), but it worsens the worst case memory access time. In contrast, a close page policy immediately precharges the bank in anticipation of an access to a different row. Therefore, the close-page policy has better memory access times for consecutive accesses to different rows, although it can no longer exploit row locality. Therefore, close-page policies are preferable for workloads with little row locality.

Table 1. Close-page DRAM timing analysis

Read Transaction	t_{RAS}+t_{RP}
Write Transaction	t_{CWD}+t_{BURST}+t_{WR}+t_{RP}+t_{RCD}

Since during a turn the active security domain is allowed to cause interference among its own memory requests, it seems reasonable, at first inspection, to allow any of these policies or a hybrid scheme to best suit the particular workload. However, the scheme as described thus far does nothing to affect the row available in sense amplifiers at the beginning of a turn. Therefore, the adversary can learn about the data access pattern of another security domain through the difference between row buffer hit and row buffer miss latencies.

This channel can be eliminated by issuing a precharge to every bank at the end of the turn. Unfortunately, contemporary DRAM chips cannot meet the power criteria necessary to issue a precharge to every bank in a sufficiently small time interval. Further, precharging only the banks which were actually accessed does not work as this implies a variable number of precharges at the end of the turn and causes yet another timing channel. The TP protection scheme thus requires a closed page policy.

3.3.4. Dead Time. With only the aforementioned restrictions, a memory transaction could be issued before the turn changes, but remain in-flight at the beginning of the next turn, possibly causing interference to memory requests from the next security domain. This interference is illustrated in Figure 9a. Therefore, an interval of time, called the dead time, is required at the end of the turn to prevent new transactions from being issued. The dead time must be long enough to complete the in-flight transactions before the turn transitions, as shown in Figure 9b. In other words, the dead time must be no less than the worst case time T_w to drain either a read or write transaction. The times required to drain either of these transactions (and precharge the bitline after the access) are shown in Table 1 in terms of DRAM timing parameters, which can be found in commercial DRAM datasheets. Based on our study of several commercial DRAM datasheets, the time to drain write transactions is usually longer than the time to drain read transactions. Therefore,

$$T_{dead} = T_w = t_{CWD} + t_{BURST} + t_{WR} + t_{RP} + t_{RCD}. \quad (1)$$

3.3.5. Refresh Timing Channel. In a conventional memory controller no transactions can be issued when a bank is being refreshed. However, when a bank that needs to be refreshed is already being accessed, the refresh is stalled

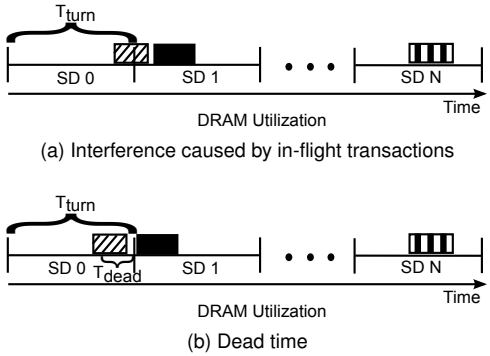

Figure 9. Dead time to remove interference from in-flight transactions.

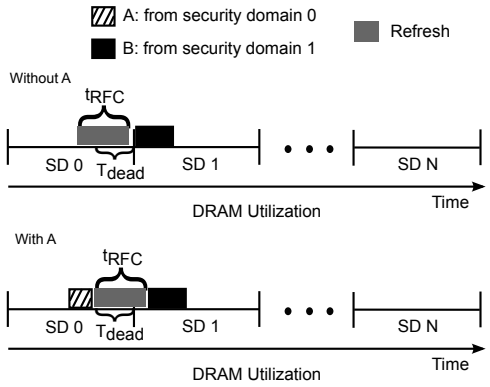

Figure 10. Interference from a stalled refresh.

until the in-flight commands to that bank are completed. This means the actual time the refresh takes place depends on the memory transactions and therefore memory access patterns and data of the active security domain. Figure 10 shows the interference caused by a stalled refresh. Without Request A, the refresh can finish before the end of SD 0's turn, and Request B can be issued as normal. However, if Request A exists and it delays the refresh, then it is possible that the refresh cannot finish until the next turn, because the time to finish a refresh, t_{RFC} is larger than T_{dead}. This indirectly delays the schedule of Request B.

This type of interference is caused by the refresh crossing the border between two consecutive turns. To solve this, the dead time can be increased to at least as long as the time to complete a refresh, t_{RFC}, plus the time required to drain any in-flight transaction, T_w. However, this is overly conservative if done unilaterally for each turn. Instead, since the originally scheduled time for each refresh is public information, only turns during which refresh is scheduled will have a dead time that is increased by t_{RFC}. This eliminates the refresh timing channel since, if the turn length is greater than $t_{RFC} + T_w$, any refresh issued during a particular turn will always finish before the end of that turn. If the turn length is less than $t_{RFC} + T_w$, the dead time is instead the entire turn length and the active security domain is blocked for its entire turn. Because there is no access from the active security domain, there is no interference between a memory access and a refresh.

3.3.6. Turn Length Tradeoff. The length of a turn affects the performance impact of temporal partitioning. There is no upper limit on the turn length, but the turn length should at least be greater than T_w to avoid a deadlock. When T_{turn} is equal to T_w, at most one request can be scheduled in one turn and it can only be scheduled at the first cycle of the turn because of the dead time. The optimal turn length depends on the workload and cache configuration among other system parameters. The tradeoffs involved are best explored by characterizing the sources of overhead in temporal partitioning.

The first source of overhead is the dead time which wastes memory bandwidth for a fixed interval at the end of each turn. The dead time comes at the end of every turn, therefore the overhead depends on the number of turns. As the turn length increases, the number of turns will reduce. As a result, the dead time overhead is less with a longer turn length. On the other hand, as the turn length increases, the maximum time a request can spend blocked in the transaction queue while its security domain is inactive also increases. Therefore, a longer turn length will be desirable when the throughput is the main concern and a shorter turn length will be desirable when the latency is important.

3.4. Hardware Implementation

To implement the temporal partitioning scheme in hardware, changes to a typical hardware memory controller are required to (a) eliminate timing channels in the queueing structure, (b) determine the active security domain, (c) determine the dead time, and (e) allow only the active security domain to issue memory requests. An close page row buffer policy must be used to remove the row buffer access timing channel.

Temporal partitioning requires the queueing structure as described in 3.3.1. This entails of separate queues and arbiters for each security domain. However, we note that this design does not necessarily require more queues than the baseline. The baseline uses a per-rank, per-bank queueing structure whereas our scheme uses a per-rank, per-security domain queueing structure. In fact, in our experimental implementation there are 8 banks and at most 4 security domains, therefore, the proposed scheme actually has fewer queues than the baseline. In order to exploit bank parallelism, this new queueing structure requires scheduling logic changes to search through the queues for the first request to an idle bank. Similar scheduling logic is needed by a conventional open page memory controller to find the first request to an open row.

Each security domain has an associated counter to check if it is the active security domain and to determine whether

or not the memory controller is currently in the dead time. The scheduler is changed to check each of these counters and accept requests only from the active security domain and outside the dead time. The length of the dead time is either the worst case transaction time, T_w, or $T_w + T_{Refresh}$ if a refresh is scheduled to happen during that turn. Since the memory controller must already calculate when refreshes take place, this only requires logic to pick between the two options.

3.5. Optimizations

3.5.1. Bank Partitioning. In TP, the memory bandwidth loss due to the dead time represents one of the largest sources of overhead. The dead time ensures that memory requests from two consecutive turns cannot interfere with each other by draining all in-flight transactions at the end of a turn before allowing any memory request from a new turn. Unfortunately, the dead time needs to be quite conservative in order to avoid interference even in the worst case where requests from two turns access the same bank.

If it can be guaranteed that requests from two consecutive turns cannot access the same bank, the dead time can be significantly reduced because in-flight transactions do not need to be drained before allowing requests from a new domain. TP can use bank partitioning among security domains or turns to guarantee this property. For example, different security domains can be mapped to different banks in the main memory. Alternatively, TP can restrict which memory banks can be used at the beginning and the end of each turn to ensure that there cannot be bank conflicts between two consecutive turns. With this optimization, the dead time can be the worst case time interval between two consecutive memory accesses to different banks. Considering the power constraint and different combinations of consecutive memory accesses to different banks, the dead time can be determined by the following equation[1]:

$$T_{dead} = max(t_{FAW} - 3 * t_{RRD}, t_{CWD} + t_{BURST} + t_{WTR}, t_{CAS} + t_{BURST} + t_{RTRS} - t_{CWD}). \quad (2)$$

For the DRAM module we used in the experiments, this new dead time is only 18 cycles compared to 46 cycles without bank partitioning.

3.5.2. Application-Aware Turn Length. In the baseline design, temporal partitioning divides the memory bandwidth evenly among security domains using the round-robin scheduling with the same turn length for all security domains. In order to distribute the memory bandwidth more effectively for a given workload mix, TP can be optimized to use a different turn length for each security domain and also schedule turns in an order that matches the workload characteristics. As long as the turn lengths and schedule are not affected by the dynamic memory demand of the security domain, temporal partitioning still ensures that there is no timing channel between security domains.

[1] This equation is an updated version of that in the original HPCA publication. It takes into account the write to read delay.

Table 2. Configuration parameters for the GEM5 and DRAMSim2 simulators.

In-Order Model			"TimingSimple"
Out of Order Model			"O3"
Number of Cores			2
Memory		2GB	667MHz
L1d / L1i	32kB	2-way	2 cycles
L2	256kB	8-way	7 cycles
L3	4MB	16-way	17 cycles

4. Evaluation

This section evaluates the security and performance of temporal partitioning through simulation studies. DRAMSim2[18] is used to model the memory controller as well as ranks, banks, and channels of the DRAM. To study the performance impact on realistic benchmarks, DRAMSim2 is integrated into the full architecture simulator, GEM5 [5].

For GEM5 experiments, we use the SPEC2006 benchmarks compiled for the ARM ISA. Each of the in-order cores uses the "TimingSimple" model in GEM5, and the out-of-order cores use the "O3" model. For each experiment, unless otherwise stated, two cores are simulated each running an independent SPEC2006 benchmark in its own security domain. Multiprogram workloads are used, because the overhead of TP comes from having two or more concurrent security domains. The cache configuration parameters such as sizes, associativity, and latencies are derived from the Intel Xeon E3-1220L, which has two cores and is similar to the CPUs used in Amazon EC2 as of late 2013. Each of the two cores has 32KB L1i and L1d caches, a local 256KB L2 cache, and a shared 4MB L3 cache. DRAMSim2 is configured to simulate a 2GB DDR3 main memory clocked at 667MHz. Table 2 summarizes the simulation infrastructure configurations. Each workload is fast forwarded through 1 billion instructions, and simulated for 100 million instructions.

4.1. Security Evaluation

Temporal partitioning eliminates the memory interference by modifying the queueing structure and the scheduling algorithm of a memory controller. To test that memory interference has been eliminated, multi-program workloads comprised of SPEC2006 benchmarks are run to record the timing of memory requests. GEM5 is used to collect memory request traces in 10 million instructions for each benchmark, then these traces are used in pairs of two (T0, T1) to study the security of a two-core system in which each core runs in a different security domain. The traces are fed into DRAMSim2 to simulate the cycle-level behavior in the memory controller and DRAM device.

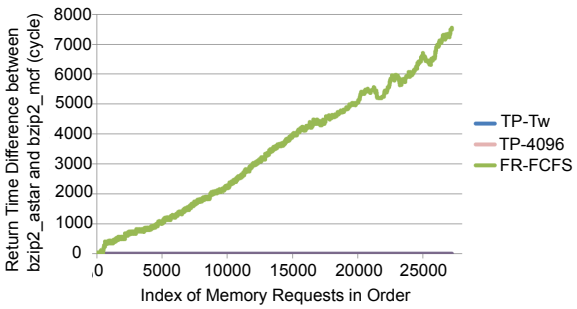

Figure 11. Memory return time difference of T0 running with difference T1s.

To verify that temporal partitioning can protect against timing channel attacks, a benchmark, T0, for one security domain is fixed and run with a different benchmark, T1. If the memory controller completely eliminates interference, the return time of each memory request in T0 should always be the same regardless of what benchmark T1 is. The results for a fixed T0 with different T1s are compared. Figure 11 shows one example of the comparison. The Y axis is the return time difference for each memory request in T0 when *bzip* is used for T0 and T1 is changed from *astar* to *mcf*. Two different turn lengths are used for TP, namely T_w and 4096 memory cycles. As can be seen, both $TP - T_w$ and $TP - 4096$ show a flat line that equals 0, meaning the timing of T0's memory requests are not affected by which benchmark T1 is. In contrast, The result for FR-FCFS shows a huge difference after T1 changes from one benchmark to another, which indicates the existence of memory interference and a timing channel. Every possible combination of benchmark pairs was compared in this way, and the results show that with temporal partitioning protection, the return time of every memory request from T0 stays the same regardless of what benchmark T1 runs.

The timing channel protection is still effective when there are more than two security domains. To test the security of increasing the number of security domains, experiments are run with four traces. T0 is kept constant and (T1, T2, T3) are changed from (*astar*, *astar*, *astar*) to (*mcf*, *mcf*, *mcf*). These two combinations were intentionally chosen because *astar* is not memory-intensive while *mcf* is memory-intensive. The results for the case where T0 is *bzip*2 are shown in Figure 12. Similar to the results for two security domains, the comparison passes for all combinations which shows that TP eliminates the memory interference for multiple security domains. This security evaluation is run for all benchmarks in SPEC2006.

4.2. Performance Evaluation

Figure 13 shows the L2 data cache misses per kilo instructions (MPKI) for the SPEC2006 benchmarks that we use in the experiments. As shown, *mcf* shows the highest memory intensity and *astar* has the lowest memory intensity.

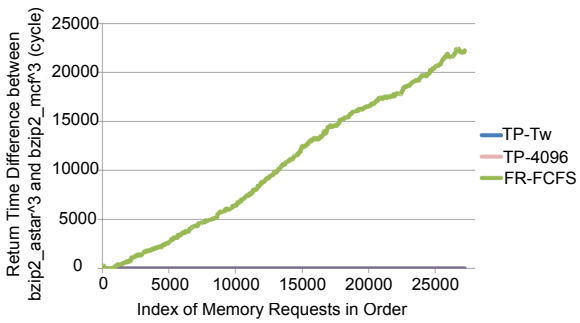

Figure 12. Memory return time difference with 4 security domains.

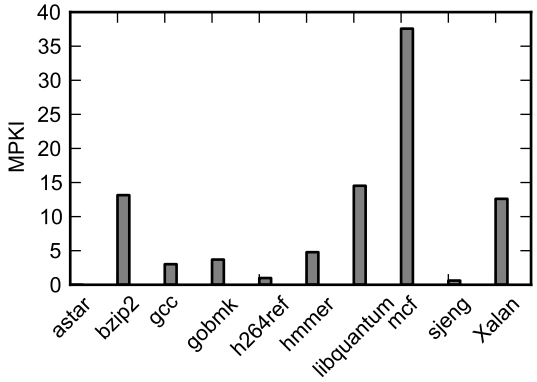

Figure 13. Memory intensity study of SPEC2006 benchmarks.

Because TP is likely to increases the memory latency and lowers the memory bandwidth in general, memory-intensive benchmarks are likely to be affected more by the overhead of TP.

4.2.1. Row Buffer Policy. Temporal partitioning cannot be easily implemented with an open page row buffer policy. Figure 14 examines the extent to which this is a drawback by comparing the performance of the baseline memory controller for the close page policy and the open page policy. Figure 14 shows the percent difference in execution time of the SPEC2006 benchmarks running with the open page policy and the close page policy over various L3 cache sizes and for in-order and out-of-order cores. For positive values, the close page policy outperforms the open page policy. Even without an L3 cache, the difference is at most 0.04% for the in-order core and 0.15% for the out-of-order core. The results suggest that the performance of the two row buffer policies are comparable for these benchmarks.

4.2.2. Performance Overhead. The static scheduling and the dead time in temporal partitioning is likely to incur performance overhead. To evaluate the performance overhead, the SPEC2006 benchmarks are run with the baseline memory controller and with TP. The experiments uses a 4MB L3 cache and the TP turn length of 64 cycles.

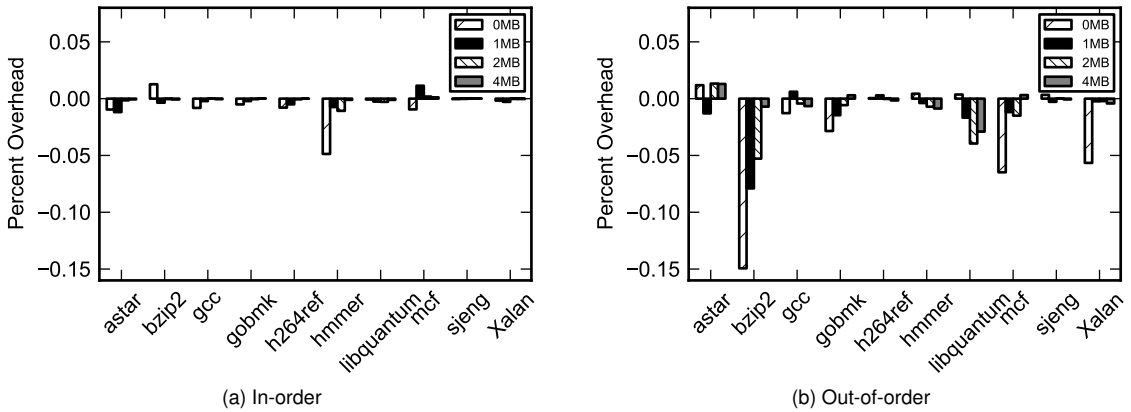

Figure 14. Effects of the row buffer policy on execution time.

Figure 15 shows the performance comparison between baseline and TP in terms of execution time using in-order and out-of-order cores. For each benchmark, multiple execution times are obtained by running that benchmark with another one, collecting results for all possible pairs. Then, the execution times with TP are normalized to the baseline and averaged. More specifically, the execution time of benchmark b_i running with benchmark b_j in another security domain, $T_{TP,b_i||b_j}$, is normalized to the execution time for the same pair in the baseline, $T_{base,b_i||b_j}$. The normalized execution time represents a slowdown due to TP. Then, the average slowdown for each benchmark is computed across all possible pairs,

$$\sum_{j=1}^{B=10} \frac{T_{TP,b_i||b_j}}{T_{base,b_i||b_j}} \cdot \frac{1}{B}, \tag{3}$$

where B is 10, the number of benchmarks used.

From the results it can be seen that the performance overheads for temporal partitioning are generally quite low. The execution time overhead for temporal partitioning is only 1.5% on average using in-order cores, and 1.4% using out-of-order cores. The benchmark, *astar*, incurs the least overhead because it has a small number of memory requests, while *mcf* has a higher overhead of 5.4% and 5.9% for in-order and out-of-order cores respectively.

Figure 16 shows the average overhead for memory latency for TP. TP increases the memory latency compared to the baseline, ranging from 60% to 150%, because the protection delays requests while its security domain is inactive. In the worst case, the memory request needs to wait in the queue while all other domains finish their turns. As a result, the delay can be as long as $(n-1)*T_{turn}$, where n is the number of security domains. The memory latency overhead can be reduced using a short turn length.

4.2.3. Sensitivity Studies. Here, we study the impact of different turn lengths and L3 cache sizes on the performance overhead of temporal partitioning. Due to space limit, we only show results for out-of-order cores. In-order core results show the same trends.

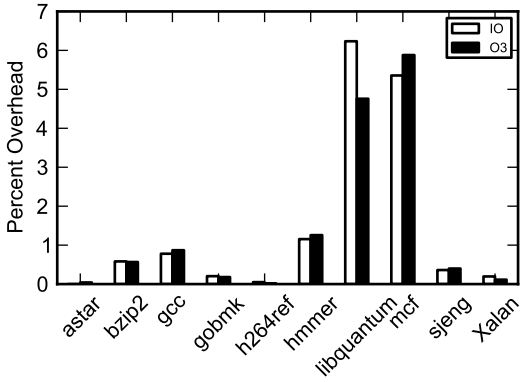

Figure 15. Execution time overhead of TP.

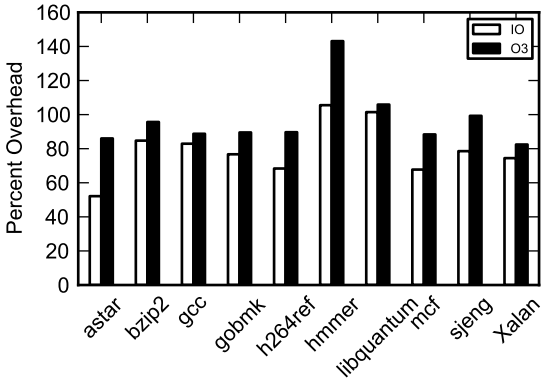

Figure 16. Memory latency overhead of TP.

There is a trade-off in selecting the turn length for temporal partitioning. Shorter turns result in lower memory latencies in general because the time that a memory request needs to wait while its security domain is inactive is reduced

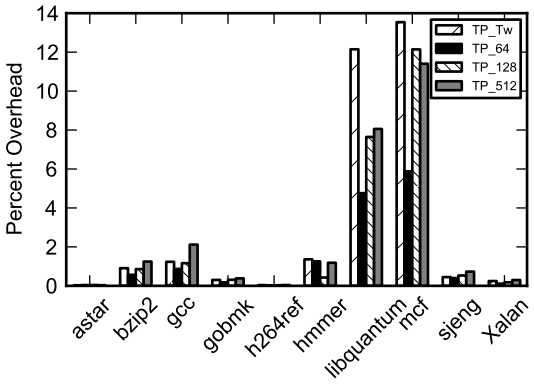

Figure 17. Effect of turn length on execution time overhead.

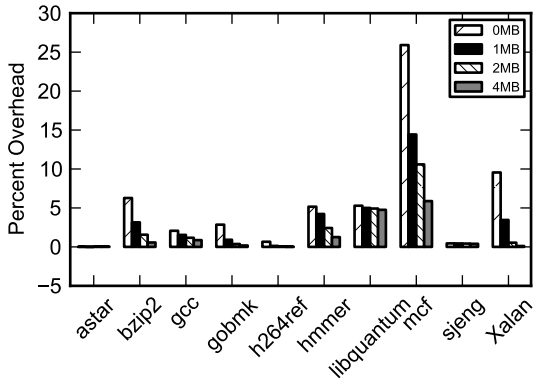

Figure 18. Effect of cache size on execution time overhead.

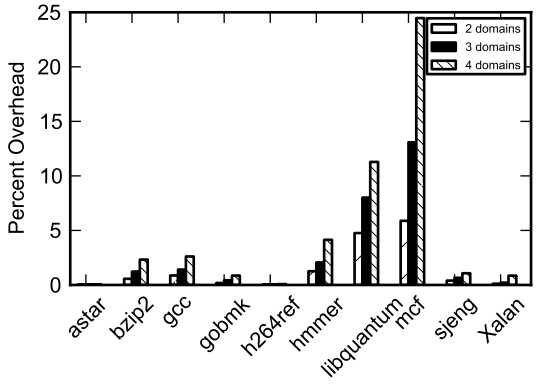

Figure 19. Effect of increasing the number of security domains.

with a shorter turn length. On the other hand, the overhead due to the dead time is lower for longer turns because it can be better amortized over a longer period. Therefore, the memory bandwidth can be better utilized with longer turns.

Figure 17 shows the effect of the turn length on the execution time overhead. The L3 cache size is fixed at 4MB. In the figure, *TP_n* indicates the timing protection using a turn length of *n*. *TP_Tw* indicates the minimum turn length. The results show that the turn length of 64 cycles provide a good trade-off between the latency and the bandwidth. In general, long turn lengths such as 128 and 512 cycles degrade the performance because they significantly increase the memory latency. However, the turn length of 64 outperforms the minimum turn length in most cases. With the minimum turn length, at most one request can be issued in each turn because only the first cycle of the turn is not in the dead time period. With a longer turn length, multiple requests can be issued in each turn utilizing bank parallelism. The increased bandwidth is especially important for memory intensive benchmarks with out-of-order cores, which are likely to have multiple in-flight memory requests at the same time.

The L3 cache size also affects the performance overhead of temporal partitioning because the cache miss-rate directly affects the memory intensity of a program. Figure 18 shows the execution time overhead of temporal partitioning for different L3 cache sizes. In this experiment, we fix the turn length to be 64 cycles and only change the L3 cache size. As shown in the figure, the performance overhead of memory-intensive benchmarks can change significantly for different L3 cache sizes. For example, *mcf* incurs 26% overhead with no L3 cache, compared to only 6% overhead with a 4MB L3 cache. On the other hand, the performance overhead of less memory-intensive benchmarks such as *astar* is much less sensitive to the L3 cache size. The results also suggest that the performance overhead of TP is quite reasonable even for small L3 caches.

4.2.4. Scalability. As the number of security domains increases, the latency overhead for requests in each security domain increases because requests need to wait for more turns for other security domains while its own domain is inactive. This implies that the overhead increases with the number of security domains. Figure 19 shows the execution time overhead of temporal partitioning as the number of security domains increases. The average overhead is shown for each benchmarks as before. However, rather than running all combinations of 3 and 4 benchmarks, the analysis uses fewer benchmark combinations. More specifically, security domains beyond the first one use the same workload.

For each benchmark, the overhead increases with the number of security domains. For 3 security domains the average execution time overhead is 2.7%. For 4 security domains this increases to 4.8%. In all cases, *mcf* incurs the highest overhead as it is the most memory-intensive benchmark. We note that the overall performance overhead is still reasonable even with the increased security domains.

4.2.5. Optimization Results. Here, we evaluate the effectiveness of the two optimization techniques in Section 3.5, namely bank partitioning (BP) and application-aware turn

length (AT), using synthetic benchmarks that simply make strided memory accesses. The synthetic benchmarks are used to study the worst-case overhead for TP with far more memory-intensive workloads compared to the SPEC benchmarks. In this study, H is a highly memory-intensive benchmark with 100 MPKI and L is a benchmark with infrequent memory requests. Each combination of these two benchmarks is run on a two-core system with a shared 4MB L3. A turn length of 64 cycles is used for the temporal partitioning scheme. For the application-aware turn length optimization, 256 cycles are allocated to H and 64 cycles are allocated to L when they are running together, otherwise each benchmark uses 64 cycles as the turn length.

The two optimizations are evaluated separately and combined, and the results are shown in Figure 20. The Y axis is the execution time overhead for the first benchmark in the pair compared to the baseline. The performance overhead results for H, shown in the first two clustered bars, clearly demonstrate that the optimizations can significantly reduce the performance overhead. The bank partitioning increases the available bandwidth for TP, and is quite effective when the workload is sensitive to the bandwidth such as in (H,H). For (H,L), the performance overhead of TP without the optimizations is as high as 150%. This is because H can only utilize at most half of the bandwidth in TP whereas it can utilize close to the full memory bandwidth in the baseline. The application-aware turn length allocates the bandwidth to match the workload characteristics. In the experiment, H can utilize four times the bandwidth of L, which is 80% of the total memory bandwidth. The performance overhead is significantly reduced to 40% with AT. Applying both optimizations can further reduce the performance overhead down to about 20% for the (H,L) combination. Figure 20 also shows the performance overhead of L in the last two clustered bars. However, the overhead is negligible at about 0.45% even without the optimizations. This is because the overhead of TP only affects memory accesses, which is quite infrequent in L.

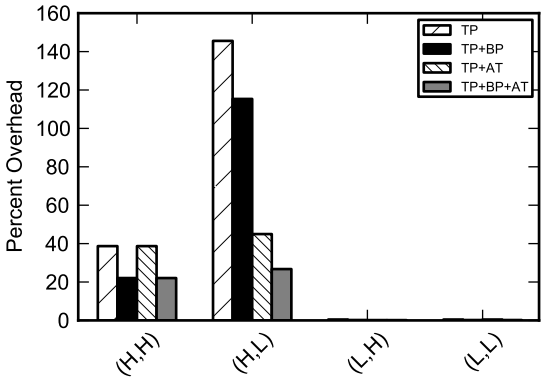

Figure 20. Performance overhead with optimizations.

5. Related Work

Microarchitecture Timing Channels Microarchitecture-level timing channels exist when microarchitecture resources are dynamically shared between attack and victim programs. Researchers have demonstrated timing channel attacks exploiting interference in various microarchitecture resources such as processing pipelines [25], branch predictors [2, 3], caches [15, 4, 1], and memory buses [29, 19]. The cache timing channel attack was also demonstrated on Amazon EC2 to recover a user password [16].

Several hardware techniques have been proposed to deal with microarchitecture timing channels. For example, protection for cache-based timing attacks [14, 9, 26, 27] and timing attacks on on-chip networks [24, 28] have been proposed recently. However, little work has been done for memory controllers. Researchers have also proposed to mitigate the timing channel attacks by injecting noise or restricting sensitive operations. For example, fuzzing a program's time has been suggested as a general countermeasure against side-channel attacks [11], and controlling the use of atomic instructions has been proposed to limit timing channels though memory bus locking [19]. This work presents the first hardware-based protection that can completely eliminate the timing channel through memory.

Verifiable Hardware Information Flow Control Timing channels represent one form of information flows. Recently, there has been significant interest in verifying the information flow properties of hardware designs, including timing channels. One such approach is to analyze and enforce information flows at a logic gate level, where implicit flows and timing channels are exposed as explicit control signals [23, 21, 12, 13, 22]. The early designs [23, 21] rely on dynamic checks with large hardware and performance overheads. The later designs extended the idea to static checks using gate-level simulations [12, 13, 22] or a language-level framework [10]. These hardware design frameworks can potentially be used to verify the non-interference property of the memory controller design in this paper. However, the design frameworks so far have only been demonstrated for simple systems with strict time multiplexing. This work investigates a memory controller design for high-end processors that support multiple concurrent security domains.

Architecture for Secure Cloud Computing Researchers have proposed a number of hardware architecture designs to enhance the security of cloud computing. These architecture techniques can significantly reduce the size of trusted software components [6, 8, 20] or even protect against physical side-channel attacks [7]. However, these secure processor designs do no consider internal timing interference among multiple concurrent program executions, which is the focus of this paper.

6. Conclusion

The shared memory in modern computing systems can cause interference among different security domains, which may be used as timing channels to extract secret information. In this paper, we identify the sources of interference existing in conventional memory controller designs. To eliminate the interference, we change the conventional per bank based queuing structure to a per security domain based queueing structure that groups the memory requests from one security domain into the same queue. The temporal partitioning scheduling algorithm is employed to time-share the memory controller among different security domains. The memory interference is shown to be eliminated by our protection scheme. In addition, we discuss the tradeoff between different turn lengths and evaluate the performance of the protection scheme, which shows negligible performance degradations.

7. Acknowledgment

This work was partially supported by the National Science Foundation under grants CNS-0746913 and CCF-0905208, the Air Force grant FA8750-11-2-0025, the Office of Naval Research grant N00014-11-1-0110, the Army Research Office grant W911NF-11-1-0082, 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, AF, ONR, ARO, or Intel.

References

[1] O. Aciiçmez. Yet another microarchitectural attack:: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, 2007.

[2] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, 2007.

[3] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 225–242, 2007.

[4] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[5] 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*, 39(2):1–7, Aug. 2011.

[6] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 16th International Symposium on*, pages 1–12, 2010.

[7] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, 2012.

[8] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual International Symposium on Microarchitecture*, MICRO-44 '11, pages 272–283, 2011.

[9] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *High Performance Computer Architecture, 2009. HPCA 2009. 15th International Symposium on*, pages 393–404, 2009.

[10] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, 2011.

[11] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2012.

[12] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical analysis of gate level information flow tracking. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 244–247, 2010.

[13] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Information flow isolation in i2c and usb. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 254–259, 2011.

[14] D. Page. Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive, report 2005/280, 2005.

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

[16] 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 Conference on Computer and Communications Security*, CCS '09, pages 199–212, 2009.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, 2000.

[18] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[19] 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.

[20] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th Conference on Computer and Communications Security*, CCS '11, pages 401–412, 2011.

[21] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42Nd Annual International Symposium on Microarchitecture*, MICRO 42, pages 493–504, 2009.

[22] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, 2011.

[23] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 109–120, 2009.

[24] Y. Wang and G. E. Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 2012 Sixth International Symposium on Networks-on-Chip*, NOCS '12, pages 142–151, 2012.

[25] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22 nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[26] 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*, ISCA '07, pages 494–505, 2007.

[27] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual International Symposium on Microarchitecture*, MICRO 41, pages 83–93, 2008.

[28] 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*, ISCA '13, pages 583–594, 2013.

[29] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.