# Memory Bandwidth Reservation in the Cloud to Avoid Information Leakage in the Memory Controller[*]

Akhila Gundu
University of Utah

Gita Sreekumar
University of Utah

Ali Shafiee
University of Utah

Seth Pugsley
University of Utah

Hardik Jain
University of Texas, Austin

Rajeev Balasubramonian
University of Utah

Mohit Tiwari
University of Texas, Austin

## ABSTRACT

Multiple virtual machines (VMs) are typically co-scheduled on cloud servers. Each VM experiences different latencies when accessing shared resources, based on contention from other VMs. This introduces timing channels between VMs that can be exploited to launch attacks by an untrusted VM. This paper focuses on trying to eliminate the timing channel in the shared memory system. Unlike prior work that implements temporal partitioning, this paper proposes and evaluates bandwidth reservation. We show that while temporal partitioning can degrade performance by 61% in an 8-core platform, bandwidth reservation only degrades performance by under 1% on average.

## Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memories—*Dynamic memory (DRAM)*
; C.2.0 [**Computer-Communication Networks**]: General—*Security and Protection*

## General Terms

Design, Performance, Security.

## Keywords

Cloud Computing, Memory Systems, Memory Controllers, Security, Timing Channels.

## 1. INTRODUCTION

Most modern computing platforms co-schedule multiple applications on shared hardware. In a cloud environment, a user application typically executes on a server with other untrusted applications. Also, an untrusted downloaded application typically executes on a user's computing device along with other trusted applications. These execution scenarios expose timing channels between the trusted and untrusted applications. By measuring delays to access shared resources, an attacking application can estimate resource usage patterns of the application being attacked. Such information leakage is then used to launch a more focused attack [2–4, 7, 17, 20, 22, 25, 29, 30]. Ristenpart et al. [20] even demonstrate one possible attack on Amazon EC2 hardware that exploits cache timing channels to recover user passwords.

Wang et al. [30] describe one possible attack that exploits memory timing channels. For an RSA decryption algorithm, the memory addresses can be configured so that the number of memory accesses is correlated with the number of 1s in the private key. The attacker can gauge the victim thread's memory traffic, estimate the number of 1s, and thus narrow the search space to determine the private key.

When two or more threads run on a server, they share many on-chip and off-chip resources, such as L1/L2/L3 caches, the on-chip network, and the memory system. Many prior works have developed solutions to reduce information leakage in caches and on-chip networks [12, 16, 24, 26–28], but only one recent paper by Wang et al. [30] has examined information leakage in a shared memory system. In this work, we start with the solution proposed by Wang et al. and show that their thread isolation policies can be greatly improved. Wang et al. propose *temporal partitioning (TP)* that allows only a single thread (or security domain) to issue requests in every time slice. We argue that it is more efficient to interleave requests from different threads. We argue that there is little to no information leakage as long as every thread takes up a pre-defined percentage of memory bandwidth.

We show that the policies of Wang et al. yield application slowdowns of 61% for eight co-scheduled threads sharing a memory controller. In comparison, our proposed policies yield a slowdown of under 1% when running eight co-scheduled threads.

## 2. BACKGROUND

In this section, we review the temporal partitioning (TP) policy of Wang et al. [30] and some DRAM basics.

Typical high-end processors can have up to four memory channels, each managed by an on-chip memory controller. When an application experiences a cache miss, a read re-

[*]This work was supported in parts by NSF grants CNS-1302663 and CNS-1314709, HP Labs, and IBM. We also thank the anonymous reviewers for many useful suggestions.

*HASP'14,* June 15 2014, Minneapolis, MN, USA
Copyright 2014 ACM 978-1-4503-2777-0/14/06 ...$15.00
http://dx.doi.org/10.1145/2611765.2611776.

quest is enqueued at the appropriate memory controller. If the cache line fetch triggers a dirty line eviction, a request is also placed in the corresponding memory controller's write queue. Reads are given higher priority when scheduling memory operations. Once the write queue reaches a high water mark, writes are drained in succession until the write queue size reaches a low water mark [9]. Similar to prior work [30], we focus our attention on reducing information leakage from a thread's read request rate.

Read requests are placed in a per-channel transaction queue. A scheduling algorithm, e.g., FR-FCFS [21], is employed to pick the best candidate for issue. This request is decomposed into its constituent memory commands (precharge, activate, column-read) that are then placed in logical per-bank in-order queues.

A memory request is delayed when there is competition for the corresponding memory bank, or for the shared address/command bus, or for the shared data bus. By measuring this delay, the attacker can estimate the level of memory activity in the victim.

To alleviate this information leakage, Wang et al. introduce Temporal Partitioning (TP). The memory controller is used exclusively by a single thread (or security domain) at a time. After a fixed time quantum (turn length [30]), it switches to a different thread. The lengths of the time quanta are determined beforehand by the operating system and/or the memory controller, based on priorities or memory demand. These lengths cannot change at run-time as a counter-measure against covert-channel attacks [30]. A suggested length for the time quantum is 96 ns.

To prevent a memory operation from spilling into the next time quantum and posing contention for the other thread, Wang et al. disallow the issue of memory operations near the end of a time quantum. This period at the end of the time quantum is referred to as the dead time (see Figure 1a). The dead time is about 65 ns and takes up most of the time quantum, i.e., only a small fraction of the time quantum is used to initiate memory requests. Wang et al. also propose a policy that allocates different threads to different banks – with such a policy, the dead time is reduced to 12 ns.

With the TP policy, every bank must be precharged at the end of every time quantum. If this is not done, then subsequent row buffer hits/misses will reveal if other threads accessed that bank during their time quanta. To avoid the need for many precharges at the end of a time quantum, Wang et al. employ a strict close-page policy.

## 3. PROPOSAL

The TP policy of Wang et al. [30] has several drawbacks that can lead to high performance degradations in memory-intensive workloads:

1. The mandatory dead time at the end of every time quantum leads to low memory bandwidth utilization. The bank partitioning approach of Wang et al. does help reduce dead time and alleviate this problem.

2. If $N$ threads are co-scheduled and $T$ is the length of each time quantum, an additional queuing delay is added for requests that arrive at the memory controller during another thread's turn. The probability of this happening is $(N-1)/N$ and such requests are delayed by an extra $(N-1) \times T/2$ cycles on average. As $N$ increases, we see that this can have a very damaging effect on memory latency and application performance.

3. The need to precharge all banks between successive time quanta, and the resulting use of a strict close-page policy can also yield sub-optimal memory performance.

To overcome these drawbacks, we propose a policy that is based on memory *bandwidth reservation (BR).*

If $N$ threads are co-scheduled, every thread is guaranteed exactly $1/N$ of the data bus bandwidth within a time quantum of length $Q$ cycles. Thus, requests from different threads are interleaved and requests are rarely forced to wait for a long time (see Figure 1b). If a thread enters a phase with low memory traffic, the memory controller inserts dummy requests on behalf of this thread to match the allocated bandwidth for that thread. If a thread enters a phase with high memory traffic, some of the requests are delayed until the next time quantum. To an external observer or attacker, every thread exhibits uniform memory traffic, thus revealing nothing about the program phase or specific key values.

The dummy requests can take many forms. To save energy, the memory controller could issue no requests and simply let the bus idle during that time. However, an attacker could later detect row buffer hits and decipher that the other threads are idling. Hence, at the very least, the dummy requests should precharge the banks even if there are no activations or data transfers.

Because threads are not perfectly isolated and because requests from different threads are being interleaved, an attacker may gather some information, but we claim that this information is of little value. For example, if the attacker observes a row buffer hit, he/she can conclude that other threads have not accessed that bank in the recent past. But by design, we know that the other threads have made a fixed number of memory accesses in the recent past. So the only information being leaked here is that the OS has coincidentally mapped recently touched pages of the victims to other banks. At worst, over time this may reveal that an application tends to favor pages that have been (coincidentally) mapped by the OS to certain banks. As another example, consider a victim thread that exhibits a high row buffer hit rate.Such a thread should pose fewer bank conflicts for other threads, thus possibly leaking some information to the attacker. However, since the attacker is forced to issue requests at a uniform rate, he/she is going to experience contention cycles in any case, i.e., the attacker has no idea if its stalls are created by demand requests from other threads or by the throttling mechanisms of the memory controller. In any case, such potential attacks can be easily thwarted by enforcing a close page policy, or by enforcing a constant row buffer hit rate policy, or by further reserving $1/N$ of every bank's bandwidth for every thread. An evaluation of performance and information leakage for this design space is left for future work.

Once a measurement time quantum ends, a new measurement time quantum of length $Q$ is considered; the starts and ends of two successive time quanta are separated by $S$ cycles, where $S < Q$. As shown in Figure 1c, this implements a coarse-grained sliding window. This is important to ensure that a thread's memory accesses are somewhat uniformly spread across the entire time quantum. If this is not done,

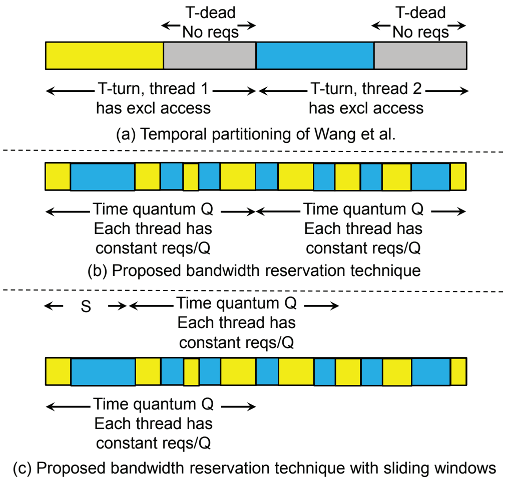

**Figure 1: Interleaving of thread requests at the memory controller for the prior work (a) and the proposed bandwidth reservation techniques (b and c).**

a non-memory-intensive thread may issue few requests at the start of the time quantum and the memory controller may insert several dummy requests at the end; this may cause some information leakage. The start and end times of the measurement quanta of each thread need not be co-ordinated. Further, the value of $S$ can be selected at random from a specified range for every new quantum. These policies make it harder for an attacker to detect if/when dummy operations are being inserted.

## 4. METHODOLOGY

For our simulations, we use Windriver Simics [1] interfaced with the USIMM memory simulator [8]. USIMM is configured to model a DDR4 memory system (bank groups, DDR4 timing parameters, etc.). Simics is used to model eight out-of-order processor cores. These eight cores share a single memory channel with four ranks. Simics and USIMM parameters are summarized in Table 1. Our default scheduler prioritizes row buffer hits, closes a row if there aren't any pending requests to that row, and uses write queue water marks to drain many writes at once [9].

We use a collection of multi-programmed workloads from SPEC2k6. Libquantum, milc, mcf, omnetpp, h264ref, GemsFDTD, leslie3d, gromacs, and soplex are run in rate mode (eight copies of the same program). Of these, h264ref and gromacs are not as memory-intensive as the others. We also execute two workloads (mix1 and mix2) that include a mix of different SPEC programs. Mix1 is a combination of perlbench, bzip2, gromacs, gobmk, hmmer, tonto, astar and sjeng. Mix2 is a combination of gromacs, gobmk, tonto, sjeng, perlbench, hmmer, mcf, soplex. SPEC programs are fast-forwarded for 50 billion instructions before detailed simulations are started. Statistics from the first 5 million simulated instructions are discarded to account for cache warm-up effects. Simulations are terminated after a million memory reads are encountered.

For now, we assume that all co-scheduled programs receive an equal share of memory bandwidth. Non-uniform allocations will benefit both the TP and BR policies.

| **Processor** | |
|---|---|
| ISA | UltraSPARC III ISA |
| CMP size and Core Freq. | 8-core, 3.2 GHz |
| ROB size per core | 64 entry |
| Fetch, Dispatch,<br>Execute, and Retire | Maximum<br>4 per cycle |
| **Cache Hierarchy** | |
| L1 I-cache | 32KB/2-way, private, 1-cycle |
| L1 D-cache | 32KB/2-way, private, 1-cycle |
| L2 Cache<br>Coherence Protocol | 4MB/64B/8-way, shared, 10-cyc<br>Snooping MESI |
| **DRAM Parameters** | |
| DRAM Frequency | 1600 Mbps |
| Channels, ranks, banks | 1 channel, 4 ranks/channel,<br>16 banks/rank |
| Write queue water marks<br>Read Q Length | 40 (high) and 20 (low)<br>32 per channel |
| DRAM chips | 4 Gb capacity |
| DRAM<br>Timing<br>Parameters<br>(DRAM cycles) | $t_{RC} = 39$, $t_{RCD} = 11$<br>$t_{RAS} = 28$, $t_{FAW} = 32$<br>$t_{WR} = 12$, $t_{RP} = 11$<br>$t_{RTRS} = 2$, $t_{CAS} = 11$<br>$t_{RTP} = 6$, $t_{DATA\_TRANS} = 4$<br>$t_{CCD\_L} = 4$, $t_{CCD\_S} = 4$<br>$t_{WTR\_L} = 6$, $t_{WTR\_S} = 6$<br>$t_{RRD\_L} = 5$, $t_{RRD\_S} = 5$<br>$t_{REFI} = 7.8\mu$s, $t_{RFC} = 260$ns |

**Table 1: Simulator and DRAM timing [11] parameters.**

## 5. RESULTS

In this initial study, we primarily focus on the impact of various policies on performance. Information leakage analyses are left for future work. Figure 2 shows the effect of the TP policy on performance, for our 8-core workloads. We assume a non-secure baseline that uses an optimized FR-FCFS scheduling policy. The bar marked "Autoprecharge" changes the scheduling policy so that every column-read and column-write include a precharge directive, i.e., a strict close-page policy. This is required by the TP policy so that there is no information leaked via row buffer hits and misses. The next bar, marked "Dead" in each graph shows the effect of introducing frequent dead times. In every 256-cycle window (80 ns), we assume that no commands can be issued in the last 172 cycles. The "Dead" model does not isolate the threads. The next bar in each graph, marked "Isolate" assumes no dead times, but only allows one thread to issue at a time. Each thread gets a turn for 256 cycles in round-robin fashion. The next bar, "TP", represents the policy of Wang et al., including a strict autoprecharge policy, thread isolation, and a dead time of 172 cycles for every 256-cycle turn. We carried out a design space exploration to confirm that a turn length of 256 cycles is optimal. The last bar, "Optimized-TP", resembles the best model of Wang et al., and assumes bank partitioning and a lower dead time of 32 cycles.

The results show that the use of a strict autoprecharge policy causes a 2.7% performance loss on average. The introduction of dead time alone causes a significant performance degradation of 22% on average. Thread isolation has

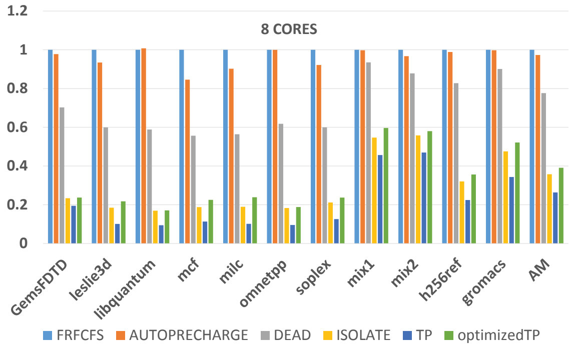

**Figure 2: Normalized IPCs for temporal partitioning (TP) in an 8-core model.**

a much more significant effect (64%) on performance because it increases wait time for most memory requests. The negative performance impact of thread isolation increases as core count is increased. The effects of each factor are roughly additive, yielding an average 74% slowdown for TP in the 8-core case. We also observe that bank partitioning provides a small performance boost, similar to what has been reported in other papers [14]. The final Optimized-TP policy is 61% worse on average than the FR-FCFS baseline.

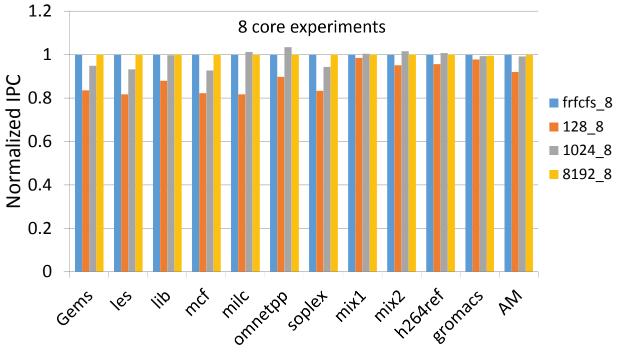

**Figure 3: Normalized IPCs for bandwidth reservation (BR) with different $Q$ values in an 8-core model.**

Next, in Figure 3, we show the effect of our bandwidth reservation policies for 8 cores. Our experiments assume that the memory controller simply idles if a thread has fewer requests than its allocation. The graph is normalized against an optimized FR-FCFS baseline and shows results for different time quanta $Q$ of 128, 1K, and 8K cycles. In these initial experiments, we assume that the value of $S$ matches $Q$. We see that a $Q$ of 128 is too restrictive in many cases, causing an 8% performance loss on average. But a $Q$ value of 8K cycles affords sufficient flexibility to the memory controller and yields a performance degradation of under 1% on average. We thus show that bandwidth reservation is far more effective than complete thread isolation.

## 6. RELATED WORK

In addition to Wang et al. [30], a few other papers have tried to eliminate timing channels in caches and on-chip networks [12, 16, 24, 26–28]. Note that techniques such as cache partitioning can isolate the cache activity of one thread from another, but it does not eliminate the memory timing channels, i.e., memory latencies of one thread are impacted by the cache miss rate of the co-scheduled thread. Martin et al. [13] thwart timing channel attacks by limiting a user's ability to take fine-grained timing measurements. Saltaformaggio et al. [22] identify potential attacks because of atomic instructions that can lock up the entire memory system; they develop solutions that require hypervisor extensions.

Information-secure memory systems can borrow designs from networking and real-time systems as well. Virtually pipelined network memory [5] (VPNM) is a secure memory system that hides information leaks among threads that contend for memory banks in network processors. VPNM normalizes memory access time to trade off latency for deterministic, high bandwidth – good for network processing, but unsuitable for standard desktop/server workloads.

Reineke et al. [19] build a DRAM controller with predictable latencies. Refreshes are broken into smaller RAS-only refresh operations to prevent disruptions from a long refresh operation. The memory controller also uses a close page policy and forcibly goes through bank groups in round-robin fashion. But a co-scheduled attacker thread can estimate the memory intensity of a victim thread because queuing delays at banks will be variable. CCSP arbitration [6] requires assigning priorities and bandwidth requirements, and regulates rate of requests to ensure bounded latency.

A few papers [10, 15, 18, 23] have used memory bandwidth reservation as a technique to aid QoS policies in datacenters. QoS policies and timing channel prevention policies differ in the following two ways: (i) QoS policies allow allocations to change based on need, and (ii) QoS policies allow a thread to steal idle resources from another thread.

## 7. CONCLUSIONS

The paper argues that bandwidth reservation yields higher performance than temporal partitioning. This is primarily because temporal partitioning introduces long wait times for memory requests that do not arrive during the thread's turn at the memory controller. Periodic dead times and a strict close page policy also contribute to temporal partitioning's high performance degradations. We qualitatively argue that the bandwidth reservation policy leads to nearly zero information leakage. This will be evaluated more rigorously in future work.

## 8. REFERENCES

[1] Wind River Simics Full System Simulator. `http://www.windriver.com/products/simics/`.

[2] O. Aciiçmez. Yet Another Microarchitectural Attack: Exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer Security Architecture*, pages 11–18, 2007.

[3] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting Secret Keys via Branch Prediction. In *Topics in Cryptology–CT-RSA 2007*, pages 225–242. Springer, 2006.

[4] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM symposium on Information, Computer and Communications Security*, pages 312–320, 2007.

[5] Agrawal, Banit and Sherwood, Timothy. High-bandwidth Network Memory System Through Virtual Pipelines. *IEEE/ACM Trans. Netw.*, 2009.

[6] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '08, 2008.

[7] D. J. Bernstein. Cache-timing Attacks on AES, 2005.

[8] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the Utah SImulated Memory Module. Technical report, University of Utah, 2012. UUCS-12-002.

[9] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads. In *Proceedings of HPCA*, 2012.

[10] R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of SIGMETRICS*, 2007.

[11] JEDEC. *JESD79-4: JEDEC Standard DDR4 SDRAM*, 2012.

[12] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Hardware-software Integrated Approaches to Defend Against Software Cache-based Side Channel Attacks. In *Proceedings of HPCA*, pages 393–404, 2009.

[13] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *Proceedings of ISCA*, 2012.

[14] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *Proceedings of MICRO*, 2011.

[15] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *Proceedings of MICRO*, 2006.

[16] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive*, 2005:280, 2005.

[17] C. Percival. Cache Missing for Fun and Profit, 2005.

[18] N. Rafique, W. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proceedings of PACT*, 2007.

[19] Reineke, Jan and Liu, Isaac and Patel, Hiren D. and Kim, Sungjun and Lee, Edward A. PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, 2011.

[20] 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*, pages 199–212, 2009.

[21] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory Access Scheduling. In *Proceedings of ISCA*, 2000.

[22] B. Saltaformaggio, D. Xu, and X. Zhang. BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels. In *Proceedings of EuroSec*, 2013.

[23] K. Sudan, S. Srinivasan, R. Balasubramonian, and R. Iyer. Optimizing Datacenter Power with Memory System Levers for Guaranteed Quality-of-Service. In *Proceedings of PACT*, 2012.

[24] Y. Wang and G. E. Suh. Efficient Timing Channel Protection for On-chip Networks. In *Proceedings of Networks on Chip (NoCS)*, pages 142–151, 2012.

[25] Z. Wang and R. B. Lee. Covert and Side Channels Due to Processor Architecture. In *Computer Security Applications Conference, 2006 (ACSAC '06)*, pages 473–482, 2006.

[26] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of ISCA*, 2007.

[27] Z. Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of MICRO*, pages 83–93, 2008.

[28] H. M. 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*, 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 *the 21st USENIX Security Symposium (Security '12)*, 2012.

[30] A. F. Yao Wang and G. E. Suh. Timing Channel Protection for a Shared Memory Controller. In *Proceedings of HPCA*, 2014.