# Defenses for Main Memory Systems using Memory Controllers

by

**Bolin Wang**

Under the guidance of

**Prof. Prashant J. Nair**

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
**BACHELOR OF APPLIED SCIENCE**

in

The Faculty of Applied Science

(in ECE)

ELEC499

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 3, 2019

# Table of Contents

# Abstract

Main memories are a key shared resource within modern computing systems. This thesis shows that memory controllers are prone to side/covert-channel vulnerabilities. The first vulnerability, called Drip-R, exploits the fact that row-buffer hits and misses incur different latency for the memory controller. The second vulnerability, called Drip-Q, leverages the read queue contention within the memory controller to fabricate differential latency. These differential latencies act as side/covert -channels and can be used to leak or receive data from other processes.

To overcome these vulnerabilities, this thesis proposes two secure and high-performance scheduling policies called Plumber-R and Plumber-Q, respectively. These policies work on the insight that request scheduling can be split into isolated epochs. We show that epochs can prevent the creation of side-channels within the memory controller and prohibit the attacker processes from leaking or receiving data using side-channels. Furthermore, within each isolated epoch, the memory requests can take advantage of row-buffer hits and improve performance. Our experiments show that, on average, Plumber-R and Plumber-Q provide 29% and 41% speedup over the prior state-of-the-art scheduling policies Close Page and Fixed Service.

## 2.1 INTRODUCTION

Modern multi-core systems enable users to concurrently use the same machine by sharing its resources. Unfortunately, sharing vital resources can introduce security vulnerabilities [1]. For instance, the memory access behaviours of one user can influence the memory latency experienced by other users. These differences in latency can be used as "side-channels" and "covert-channels" to leak data [2,3]. Recently, attacks like Spectre [4], Meltdown [5], and Foreshadow [6] have exploited side-channels within caches. Furthermore, researchers from academia and the industry have suggested hardware-software mitigation [7, 8, 9, 10, 11]. However, even if we mitigate the side-channels within caches, other levels of the memory-hierarchy can provide avenues for creating side/covert-channels [1]. This thesis aims to develop secure and high-performance memory scheduling algorithms that mitigate these side-channels.

Main memory systems typically consist of modules that use high-capacity Dynamic Random Access Memories (DRAM). Each module is managed by a memory controller. Unfortunately, while DRAM systems provide very large capacities, they also have high access latencies [12]. For instance, an access to an arbitrary Address-A ($\text{RD}_A$) can take nearly 40ns [13]. Memory modules try to leverage the spatial locality of workloads to reduce access latency. On an access, memory modules typically prefetch a row of DRAM cells (4KB-16KB long) onto a row-buffer [14]. If the workload has high spatial locality, only the first memory access, $\text{RD}_A$, would encounter a latency of 40ns. Subsequent accesses to neighboring addresses, like $\text{RD}_{A+1}$, would only encounter a latency of nearly 13ns (row-buffer hit). To ensure a high row-buffer hit-rate, the memory controller maintains read and write queues to capture accesses to neighboring addresses.

Unfortunately, row-buffers and read-queues may not always reduce the access latency. For example, an access to an address that is not present within a row-

[1]This thesis targets both side-channels and covert-channels in its threat model. For simplicity, we only use the term side-channel to describe one of these threats in detail. The ideas in this thesis are broadly applicable to enable or mitigate both side-channels and covert-channels

buffer will require that the old DRAM row within this row-buffer be closed and a new DRAM row to be opened (row-buffer miss). Therefore, this access would encounter an additional latency of 40ns. Furthermore, each pending request in the read-queue would also experience an additional queuing delay. An attacker can exploit the difference in latencies between row-buffer hits and misses. Similarly, an attacker can also exploit the difference in latencies due to queuing delays. These latency differences can be used to create side-channels.

Figure 2.1(a) shows two side-channel vulnerabilities Drip-R and Drip-Q, that exploit differences in row-buffer hit/miss latencies and read-queue contentions respectively. To prevent Drip-R vulnerability, prior work proposes the "Close-Page" (CP) policy that closes row-buffers after every access. To prevent Drip-Q vulnerability, prior work recommends maintaining a constant memory access rate, called "Fixed-Service" (FS) policy, by actively inserting fake requests. Figure 2.1(b) shows the normalized slowdown of these policies with respect to a "not-secure" first-ready first-come first-serve (FRFCFS) policy. On average, these policies show a slowdown of 34% and 60% respectively. To address these performance concerns while providing security this thesis proposes the Plumber-R and Plumber-Q scheduling policies.

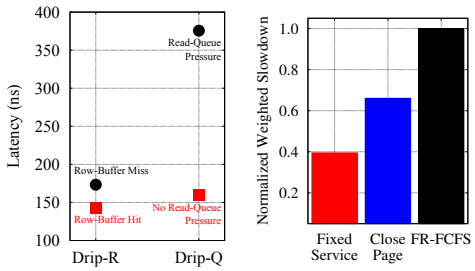


Figure 2.1: (a) Shows the differential latency due row-conflict (Drip-R) and read queue contention vulnerabilities (Drip-Q). (b) Shows two secure prior works: Close Page prevents against Drip-R, Fixed Service protects against Drip-Q. As compared to an unsecured baseline of FR-FCFS scheduling policy, the secure schedulers are 35%-60% slower. The goal of this thesis is to propose a secure high-performance scheduling policy.

The Plumber-R and Plumber-Q policies maximize row-buffer hit-rates while protecting against Drip-R and Drip-Q.

**A) Drip-R Vulnerability:** The Drip-R vulnerability tries to cause row-buffer conflicts and transmit information via side-channels. This can be orchestrated even if the malicious process that creates the side-channel and the attacker do not share their address space. The attacker would simply keep a row-buffer open and wait for the malicious process to either close the opened row-buffer or keep the row-buffer open. After some time, the attacker would probe another location from the opened row-buffer and measure the access latency. If the row-buffer is open, then attacker would see a lower latency, otherwise the attacker would see a higher latency.

**B) Plumber-R Scheduling Policy:** DRAM memory modules have high access latencies. As such, the read and write queues in the memory controller are usually filled at a rate faster than they can be drained. Therefore it is likely that, even before the completion of the first memory request within the read queue, the read queue fills up to have several pending memory requests. Plumber-R avoids Drip-R by creating time-epochs within the read-queue. A time-epoch begins when the first access is issued. The time-epoch takes a snapshot of the read-queue as soon as they begin. This snapshot captures the total number of requests present in the read-queue while the first access was being issued. Plumber-R issues request(s) only within its current time-epoch. Plumber-R ignores all requests which lie outside its current time-epoch (possibly pending in the read-queue). The time-epoch ends *only* when the last request in the snapshot has completed accessing the memory.

For instance, if at time $t_0$, the read queue has *only* 1 request, then the epoch begins at time $t_0$ and ends when this request is serviced (say at time $t_{39}$). Let us assume that the read queue is filled with 10 pending requests when the time-epoch ends (at $t_{39}$). Plumber-R waits for the current epoch to end at $t_{39}$. It then closes all opened row-buffers and creates a new epoch that starts at time $t_{40}$. This new time-epoch ends only when all the 10 pending requests are serviced. *All requests*

*within the same time-epoch are rescheduled by Plumber-R to maximize row-buffer hit-rates and improve performance.*

The attacker must issue two memory requests in order to use the Drip-R vulnerability. In the first request, the attacker opens a row-buffer. In the second request, the attacker checks if the opened row-buffer is still open or not. For the Plumber-R policy, the two memory requests of the attacker must not be issued in two different time-epochs. This is because, while switching between these two time-epochs, all the row-buffers would be closed by Plumber-R. Therefore, the attacker is forced to issue all its memory requests in the same time-epoch. Unfortunately, in this scenario, these two requests will be issued back-to-back by the Plumber-R scheduler (to maximize row-buffer hit-rates). Due to this, the attacker will always see a low latency if both its requests lie in the same time-epoch.

**C) Drip-Q Vulnerability:** An attacker can also create a side-channel using read-queue contention. For instance, a naive user (victim) can execute a malicious process that overwhelms the read queue with multiple read requests. This contention can be probed by the attacker by simply issuing one additional read request to the read queue and measuring the queuing delay. We call this side-channel vulnerability as Drip-Q. Like Drip-R, Drip-Q does not require the attacker and the victim users to share address space.

**D) Plumber-Q Scheduling Policy:** Plumber-Q is a high-performance scheduling policy that is designed to prevent the Drip-Q vulnerability. As Drip-Q relies on the number of pending memory requests in the read-queue, Plumber-Q redefines the time-epoch from Plumber-R to be independent of the number of requests. Therefore, the time-epoch for Plumber-Q is chosen as an arbitrary slot of time. The length of the time-slots are dynamically tuned to maximize the row-buffer hit-rate while maintaining a high IPC.

Plumber-Q allots the same amount of time for each process in the read-queue. Like Plumber-R, *all requests within the same time-epoch of Plumber-Q are resched-*

*uled to maximize row-buffer hit-rates and improve performance.* If all requests within the time-epoch are completed before the time-slot ends, then the current process (in-charge of this time-slot) simply closes all opened row-buffers and remains idle. Once the time-epoch ends, the memory controller moves to the next process in a round-robin manner. Therefore, all processes experience the same queuing delay irrespective of the number per-process requests in the read queue. Therefore, Plumber-Q prevents the creation of side/covert-channels using Drip-Q. Furthermore, as Plumber-Q fundamentally uses time-epochs, it can also prevent against side/covert-channels using Drip-R.

Plumber-R and Plumber-Q scheduling policies provide an average Normalized Weighted Speedup of 29% and 41% over prior state-of-the-art policies. These schedulers improve security by preventing the creation of side/covert-channels.

## 2.2 BACKGROUND AND MOTIVATION

We provide a brief background on the organization of main memory systems and the threat model.

*Main Memory System: Organization*

Main memory systems are composed of DRAM modules that operate on independent buses called memory channels. Each channel is managed by a memory controller. Typically, memory channel consists of multiple ranks. Each rank consists of several banks and each bank consists of multiple 4KB to 16KB sized rows of DRAM cells. Each bank also contains a 4KB to 16KB sized row-buffer that stores the most recently accessed row of DRAM cells [15]. Each memory request typically addresses only 64-Bytes of data. Therefore, a large row-buffer can hold several consecutive 64-Byte data elements (64 to 256 data elements). Accessing addresses within a row-buffer is typically 4x-5x faster than accessing addresses that are not within the row-buffer [14, 16]. A memory controller issues read and write requests and helps

maintain the timing constraints. Typically, memory controllers maintain separate read and write queues. To reduce access latency, the memory controller tries to maximize row-buffer hit-rate by issuing requests to open row-buffers.

*Threat Model*

We focus on threat scenarios in which processes from distrustful security domains are simultaneously executed on a hardware system with a trusted Operating System or Hypervisor [17]. Our threat model assumes that, except the memory controller, the hardware system is fully trusted and secure. These threat environments can occur in shared data-centers, workstations, or the cloud. The users in these environments can be executing on a bare-metal machine or simply using co-located Virtual Machines (VM). Adversarial processes can actively try to leak sensitive data from a victim user to an attacking user using a side-channel. A malignant process, in the pretext of performing useful tasks, can also leak sensitive data to the attacking user. This type of channel is called a covert-channel. While this thesis discusses side-channel mitigation, our threat model and our proposal also mitigates against covert-channels. This is because, our aim is to relieve the user from classifying the trustworthiness of applications (including legacy applications) and therefore our threat model categorizes all processes with the same threat perception.

The differential latency between a row-buffer hit and a row-buffer miss within a DRAM system can be used by a malignant process to potentially leak sensitive data. Prior work, DRAMA [18], uses hardware and software probing to determine the address-mapping (bank and row information) and leak data using row-buffer conflicts on targeted banks. To the best of our knowledge, there is no prior work that showcases this side/covert-channel vulnerability without somehow determining the address-mapping a priori.

Alternately, rather than relying on row-buffer conflicts, a malignant process can also cause read queue contention to leak data. To the best of our knowledge, no

prior work has showcased this hypothetical side/covert-channel vulnerability on a memory controller. To orchestrate this, a victim user executing such a malignant process should be able to overwhelm the read queues of the memory controllers based on the value of the data it wants to leak. The attacking user can then issue a single read request to probe the read queue. A filled read queue would offer high contention and thereby showcase a much longer read latency as compared to an empty read queue.

*Scheduling Policies: Secure vs Not Secure*

Prior work such as "First-Ready First Come First Serve" (FR-FCFS) schedulers scan the read and write queues for requests to open row-buffers [19]. The FR-FCFS scheduler issues these requests one after another. While this reduces the latency of memory requests, it does not mitigate side/covert-channels that can be created using row-buffer conflicts.

A secure scheduling policy can be designed to close the row-buffer as soon as it open. This is called a "Close-Page" (CP) scheduling policy and prevents the creation of row-buffer conflict side/covert-channels [20]. To prevent potential read-queue contention side-channels, prior work has proposed using a "Fixed Service" (FS) scheduling policy [17]. In the FS policy, requests from each core are scheduled in a round-robin fashion at the memory controller. If a core has no request to be issued, then the memory controller issues fake requests. This ensures that all cores experience the same contention. Unfortunately, the FS policy eliminates row-buffer locality while also consuming additional bandwidth for fake requests. Thus prior work have primarily focused on policies that eliminate row-buffer hit rates to provide security against the creation of side-channels.

Figure 2.2 shows the read-queue delay in processor cycles. Similar to prior work, we simulate a system with 8-cores and one main memory channel. We evaluate against a total of 43 SPEC2006 and MIX workloads. On average, the read-queue

delay of Fixed Service scheduling policy is 4600 cycles. On the other hand, Close Page reduces the average read-queue latency to 1300 cycles. The "not secure" FR-FCFS policy outperforms these secure schedulers by showcasing a read-queue delay of 500 cycles. Therefore, secure schedulers within memory controllers usually pay significant performance costs.

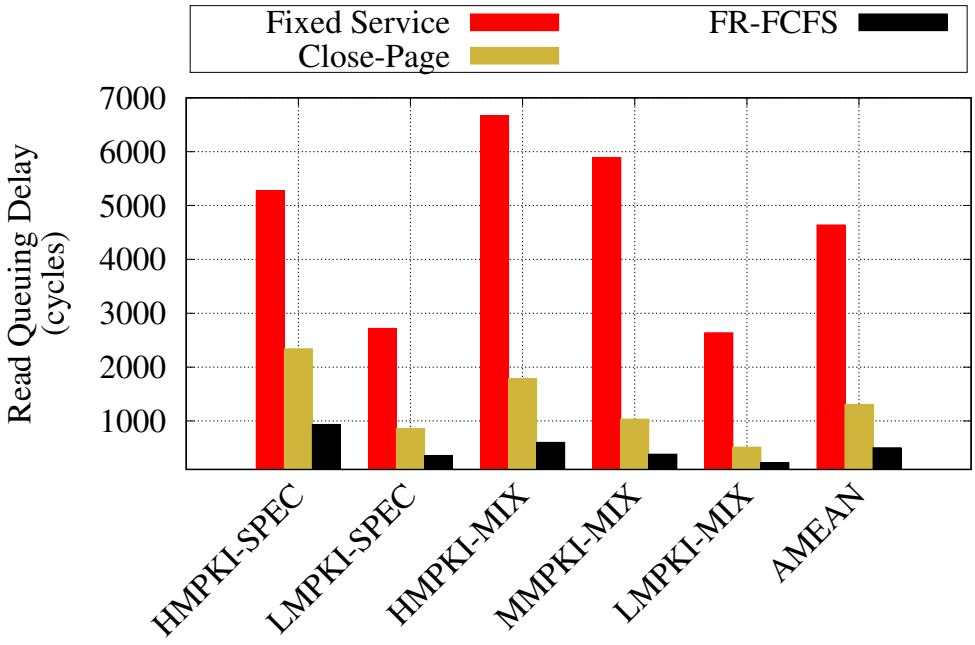


Figure 2.2: The effective read queuing delay for Fixed Service, Close Page and FR-FCFS schedulers. As we move from "not secure" schedulers to "secure schedulers, the queuing delay increases. On average, the read queuing delay increases from 500 cycles to 4700 cycles.

*Motivation: Potential Row-Buffer Hit-Rates*

Figure 2.3 shows the potential row-buffer hit-rates by simulating a single-core system with a single channel executing a single copy of SPEC2006 benchmark suite. On average, if we can design a secure scheduling policy that captures row-buffer locality then we can potentially improve the hit-rate to 62%.

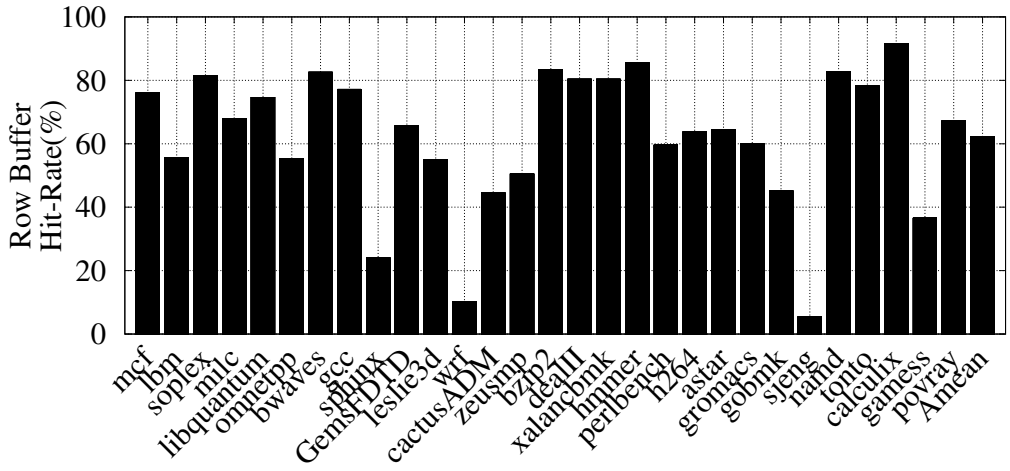


Figure 2.3: The effective row-buffer hit-rate SPEC2006 benchmarks when they are run on a single-channel in isolation using FR-FCFS policy. On average, we observe a relatively high row-buffer hit-rate of 62% and potentially we can design a secure scheduler that improves the row-buffer hit-rate.

## 2.3 DESIGN: THE PLUMBER SCHEDULING POLICIES

This section describes insights in the design of the Plumber-R and Plumber-Q scheduling policies.

*Plumber-R: A scheduler that tackles Drip-R vulnerability*

The Plumber-R scheduler tackles the Drip-R vulnerability by ensuring that the row-buffer conflict information is not transmitted. To this end, Plumber-R uses queue-epochs and uses a "marker" to identify the end of a queue-epoch.

*Begin a Queue-Epoch using a Marker*

The Marker is a hardware register that is implemented within the memory controller to track the last request of the current queue-epoch. Figure 2.4(a) shows the implementation of a marker within the memory controller. By default, the head of the read/write queue is the first request within the queue-epoch (e.g. Addr-A

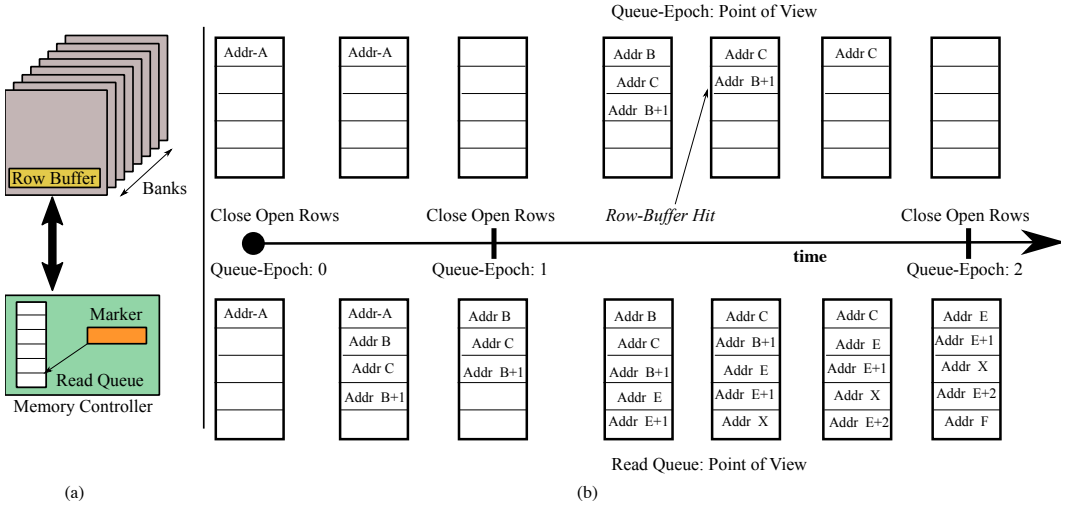


Figure 2.4: (a) A memory controller implementing the queue-epoch scheduler along with a marker to identify the last request for the queue-epoch. (b) A timeline of the working of queue-epoch. Within the epoch window, the queue-epoch tries to maximize hit-rates and improves performance. Across epoch windows, the queue-epoch closes all row-buffers and provides security.

for queue-epoch:0). To implement a queue-epoch marker, say for a 96 entry read and a 96 entry write queue, the memory controller requires two 7-bit registers. As 7-bits can address up to 128 potential marker locations within these queues. The first 7-bit register tracks the location of the queue-epoch marker request within the read queue. The second 7-bit register tracks the location of the queue-epoch marker within the write-queue. Thereafter, the memory controller will try to issue the first permissible request between the head of the queue and the request signifying the queue-epoch marker.

*High-Performance Scheduling within a Queue-Epoch*

Figure 2.4(b) shows how Plumber-R improves its performance. After issuing the first request, the memory controller switches its scheduling mode to the FRFCFS policy. For instance, after queue-epoch:1, the memory controller has multiple choice of requests. Before issuing each request, the memory controller consults the Marker

to verify that the current queue-epoch has not ended. If the queue-epoch has not ended, the memory controller prioritizes requests to open row-buffers and tries to maximize the row-buffer hits to reduce the average access latency. For instance, in queue-epoch:1, Addr-B and Addr-(B+1) were scheduled one after another. If the memory controller is unable to find any requests to an open row-buffer, it then issues requests that potentially would have to close old DRAM rows and open new DRAM rows into the row-buffers (e.g. Addr-C).

*Securely Ending a Queue-Epoch*

Eventually, the Marker will notify the memory controller that the present request denotes the end of the current queue-epoch. The memory controller then issues the current request and thereafter closes all open row-buffers in the channel. Once all row-buffers are successfully closed, the Plumber-R scheduler notifies the Marker that Plumber-R will begin of a new queue-epoch now. As all row-buffers are closed before the new queue-epoch, requests in the new queue-epoch are unaware of the previous state of the row-buffers (open or closed).

*Plumber-R: Security Analysis*

The flowchart in Figure 2.5 can be used to explain the security features of the Plumber-R policy. If an malicious user (attacker) wants to create side/covert-channels using Drip-R, then the attacker has three possible options. In the first option, the attacker inserts a request into the read-queue in any single queue-epoch. Thereafter, in the same queue-epoch the malignant process in the victim userspace inserts another request into the read queue to close the opened row-buffer. Subsequently, in the same queue-epoch, the attacker then inserts another read request into the read queue to check if the older row-buffer is open or not. Unfortunately for the attacker, as all three requests are in the same queue-epoch, the two requests to the same row-buffer by the attacker are issued one after another by Plumber-R. In

this scenario, the second request of the attacker always experiences a row-buffer hit. As the attacker request was the first to enter the read-queue, it lies at the beginning of the read queue. Hence, the pending request from the victim user is issued only after the pending requests by the attacker are complete.

In the second option, the attacker inserts a request into the read-queue in a queue-epoch. In the same queue-epoch, a malignant process in the victim userspace inserts another read request to close the row-buffer that was opened by the attacker. In the next queue-epoch, the attacker inserts a probing read request into the read queue to check if the row-buffer is still open. Fortunately, as all the previous requests by the attacker and the victim user were in the previous queue-epoch, the second request by the attacker will experience a row-buffer miss latency. This is because, the Plumber-R policy guarantees that it will close all open row-buffers between epochs.

In the third option, the attacker inserts a request into the read-queue in any queue-epoch. Thereafter, in the subsequent queue-epoch, the malignant process in the victim userspace inserts another request into the read queue. As the queue-epochs have changed, the row-buffers that were open by the attacker is now closed. Therefore, as row-buffers closures ensure that the side/covert-channels was destroyed even though the attacker has not issued its second request. This is because, Plumber-R closes all open row-buffers between epochs.

*Plumber-R: Row-Buffer Hit-Rate*

Figure 2.9 shows the row-buffer hit-rate of Plumber-R scheduling policy. To find out the hit-rate for each workload, we use a single-core configuration that connects via a single channel to the memory system. This ensures that the accessing core has complete memory bandwidth. On average, in spite of providing security against side-channels that target row-conflicts, Plumber-R still shows a row-buffer hit-rate of 23%.

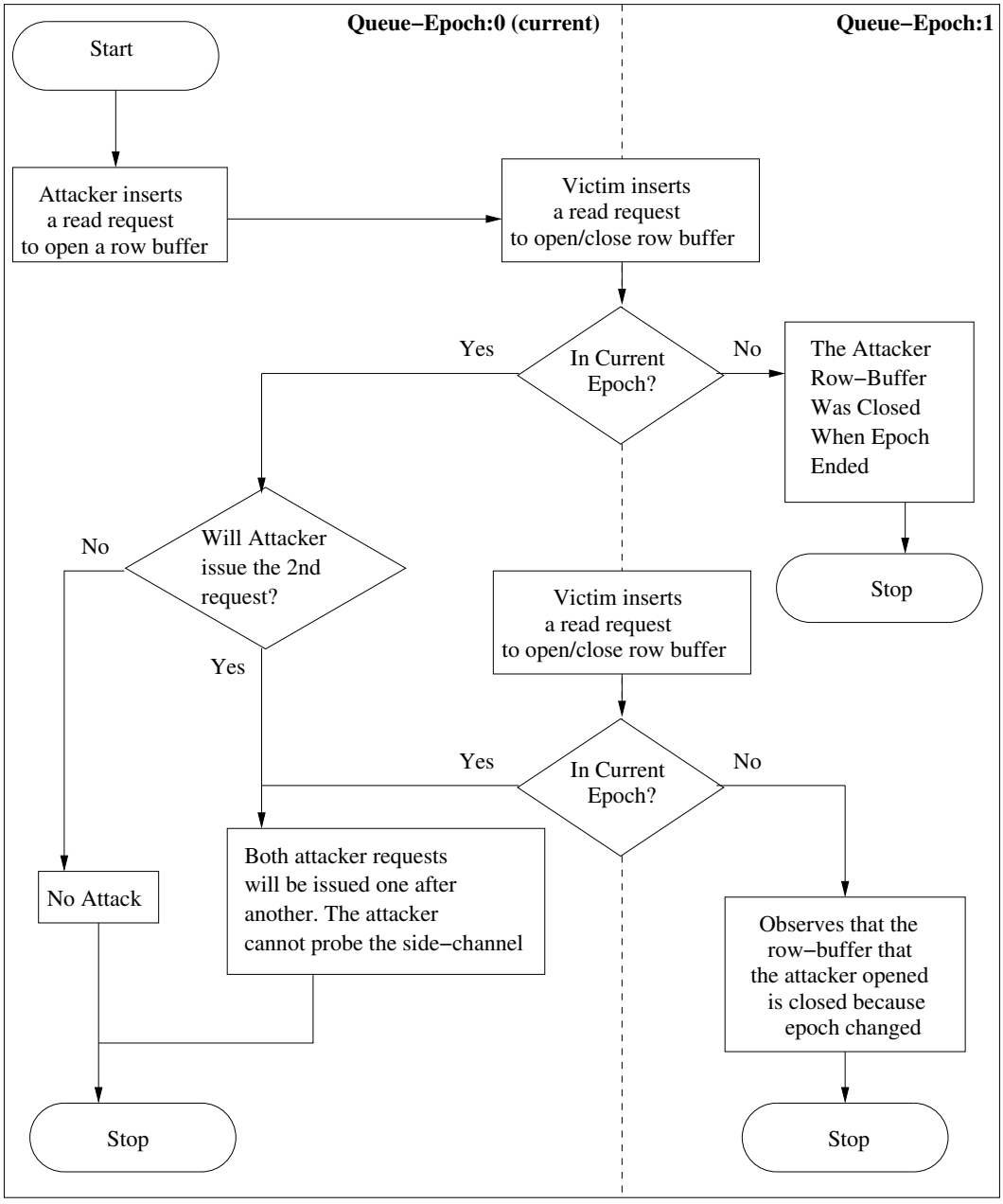


Figure 2.5: The flowchart of the mitigations of Plumber-R against Drip-R vulnerabilities. Plumber-R prevents the side-channels from being formed and thereby protects the victim user from leaking data via the memory controller.

*Plumber-R: Queue-Epoch Size vs Read-Queue Occupancy*

Figure 2.7 shows the read queue occupancy vs the queue-epoch size for the *mcf* workload. The workload, mcf, has a high memory intensity and its memory intensity

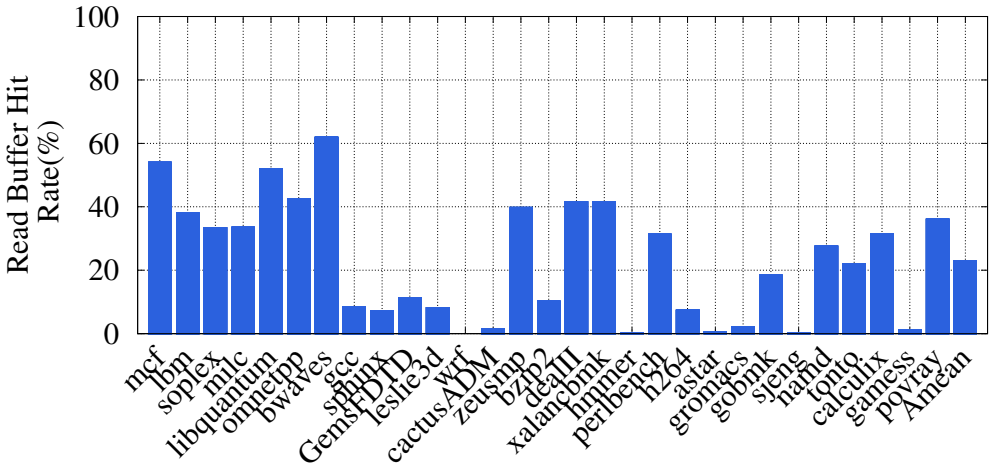


Figure 2.6: The effective row-buffer hit-rate SPEC2006 benchmarks when they are run on a single-channel in isolation using Plumber-R policy. As compared to other secure scheduling policies that tend to show 0% hit-rate, we observe an average row-buffer hit-rate of 23%.

varies rapidly over time. We observe that the number of requests in the queue-epoch closely tracks the read queue occupancy. Figure 2.7 helps provide an intuition as to why the row-buffer hit-rate is high while using the Plumber-R scheduling policy.

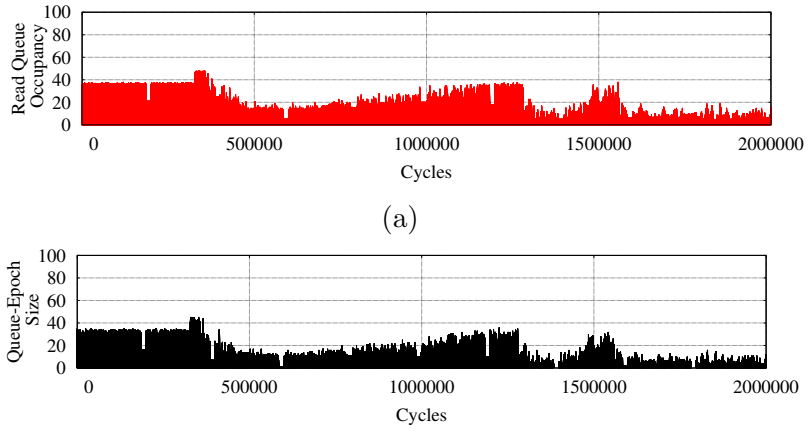


Figure 2.7: (a) Shows the read-queue occupancy for the *mcf* benchmark from the SPEC2006 suite. Even over an interval of 2-Million processor cycles, the occupancy of the read-queue varies between 50 entries and 10 entries (b) Shows the trend for the queue-epoch size for the *mcf* benchmark. Over an interval of 2-Million processor cycles, the queue-epoch size closely tracks and follows the read-occupancy values. Therefore, if the benchmark has a high row-buffer hit-rate, the queue-epoch sizes will ensure that the hit-rate is maximized.

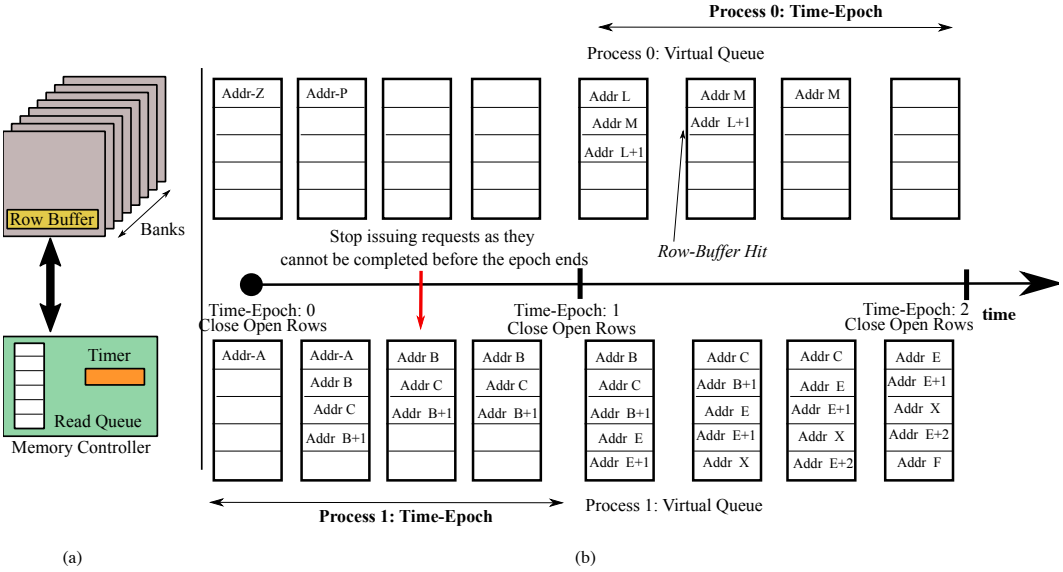


Figure 2.8: (a) A memory controller implementing the time-epoch scheduler along with a timer to identify the completion time of time-epoch. (b) A timeline of the working of time-epoch. Within the epoch window, the time-epoch tries to maximize hit-rates and improves performance at a process-level. Across epoch windows, the time-epoch closes all row-buffers and provides isolation between processes.

*Plumber-Q: A scheduler that tackles Drip-Q vulnerability*

The Plumber-Q scheduler tackles the Drip-Q vulnerability by ensuring that read queue contention information is not transmitted between processes. To this end, Plumber-Q uses time-epochs of arbitrary lengths of time and uses an "epoch timer" to identify the end of the epoch.

*Evolving a Queue-Epoch as a Time-Epoch*

Unlike Drip-R which uses row-buffer conflicts to leak sensitive data, Drip-Q uses read-queue contentions. Therefore, Plumber-Q should try to maintain the same queuing delay for all requests across processes so as to leak no data to any probing process. Plumber-Q enables this by re-defining the epoch into arbitrary units of time called time-epochs. The key point is that the length of the time-epochs do not

reflect the number of requests in the read queue. However, the time-epoch slots are varied dynamically to improve performance.

As shown in Figure 2.8(b) Plumber-Q isolates processes by maintaining a separate virtual read and write queues for each process. These read and write queues are accessed in a round-robin fashion such that each process gets its fair share of memory bandwidth. The length of time a virtual queue can exclusive rights to use the memory bandwidth is determined by the time-epoch. After the time-epoch expires, the current process must relieve the memory bandwidth, close all open row-buffers. A new process now gets exclusive rights for the memory bandwidth for a duration of time-epoch. The key invariant for Plumber-Q is that processes can use the memory bandwidth only during their epoch-slots.

The time duration of the epoch slots are varied as per the row-buffer hit-rates. However, if the memory controller wants to update the value of the time-epoch, it must wait until the round-robin process across all cores is complete. This ensures that no one process can see a drop in contention or higher priority. Therefore, the time-epoch changes are only reflected when the processes in the first core regain exclusive access to the memory bandwidth for their virtual queues. At this stage, the length of time for the time-epoch is set based on the row-buffer hit-rate. In our studies, we empirically determined that we can increase the length of time for the time-epoch by 'x' nanoseconds, if the row-buffer hit-rate also increases by x%. Therefore, the maximum length of time for the time-epoch slots can be 200ns.

*Begin a Time-Epoch in Plumber-Q: Epoch Timer*

As shown in Figure 2.8(a) The epoch timer is implemented within the memory controller and tracks the ending time of the current time-epoch for the virtual queue of a process. By default, the head of the virtual read/write queue at the beginning of the epoch-slot is the first request of this time-epoch. To keep track of the time left, each process per core is equipped with an epoch timer. When the process

is idle, this timer may be stored in its virtual memory and can be paged into the memory controller by the Operating System when the process starts executing. The number of bits in the timer depends on the maximum length of the epoch slot. For our studies, our initial epoch slot can be 100ns and it can extend to a maximum of 200ns. Therefore, for each virtual read and write queue, the memory controller requires an 8-bit register for each process.

*High-Performance Scheduling within a Time-Epoch*

After issuing the first request, the memory controller switches its scheduling mode to the FRFCFS policy. Before issuing each request, the memory controller consults the epoch-timer to verify that the current epoch slot has not ended. If the time-epoch has not ended, for subsequent requests, the memory controller prioritizes requests to open row-buffers from within its own virtual queues. If the memory controller is unable to find any requests to an open row-buffer, it then issues other requests from its virtual queue.

*Plumber-Q: Security Analysis*

If an malicious user (attacker) wants to create side/covert-channels using Drip-Q, then the attacker has only one option. During its turn in the round robin scheduling for Plumber-Q scheduler must establish a side-channel or a covert channel with a malicious process that is run on the victim userspace. Therefore, the attacker can design the malignant process to "flood" the memory controller with memory requests if it wants to transmit a binary "1". Thereafter, the attacker will try to insert a probing request at memory controller and determine the read-queue latency.

Fortunately, such an action would be thwarted by the Plumber-Q scheduler. Even during the time-epoch of the malignant process, the Plumber-Q scheduler requires that the process finish executing their memory requests before the "epoch timer" expires. Therefore, once the malignant process issues the allotted set of

requests, the memory controller would simply stall this process. Plumber-Q would not allow additional requests to be issued until the previous requests are returned from the memory. Furthermore, at the end of the epoch, the Plumber-Q scheduler closes all open row-buffers.

*Plumber-Q: Row-Buffer Hit-Rate*

Figure 2.9 shows the row-buffer hit-rate of Plumber-Q scheduling policy. To find out the hit-rate for each workload, we use a single-core configuration that connects via a single channel to the memory system. This ensures that the accessing core has complete memory bandwidth. On average, in spite of providing security against side-channels that target read contentions, Plumber-Q shows a row-buffer hit-rate of 32%.

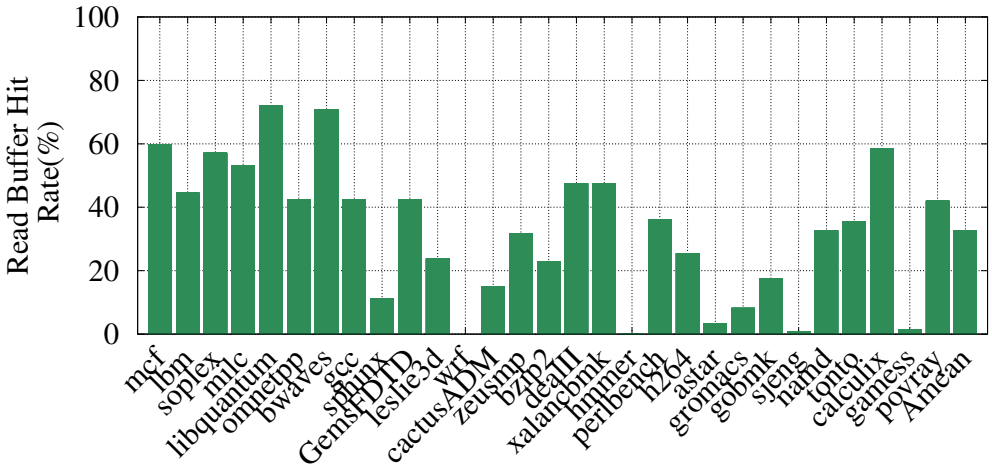


Figure 2.9: The effective row-buffer hit-rate SPEC2006 benchmarks when they are run on a single-channel in isolation using Plumber-Q policy. As compared to other secure scheduling policies that tend to show 0% hit-rate, we observe an average row-buffer hit-rate of 32%.

## 2.4 EXPERIMENTAL METHODOLOGY

To evaluate the performance benefits of Plumber-R and Plumber-Q scheduling policies, we develop a trace-based simulator based on the USIMM [21]. USIMM provides a detailed memory system model and was used for the memory scheduling championship. We extended USIMM to model the processor core and construct a detailed cache hierarchy. Our processor model supports the out-of-order (OoO) execution with 4-wide issue. The baseline system configuration is described in Table 2.1. We also implement Close-Page and Fixed Service secure schedulers [17]. We compute the weighted IPC for each workload. To do this we compute the IPC of the workload on a single-core system while running FR-FCFS and using all its resources. For power consumption, we incorporate the internal power calculator of USIMM [21].

Table 2.1: Baseline System Configuration

| | |
|---|---|
| Number of cores (OoO) | 8 |
| Processor clock speed | 3.2GHz |
| Issue width | 4 |
| Last Level Cache (Shared) | 4MB, 8-Way, 64B lines |
| LLC Tag Access Latency | 35 cycles |
| LLC Data Access latency | 5 cycles |
| Memory | DDR3-800 |
| Memory channels | 1 |
| Ranks per channel | 2 |
| Banks | 8 |
| Rows per bank | 64K |
| Columns (cache lines) per row | 128 |
| DRAM Access Timings: $T_{RCD}$-$T_{RP}$-$T_{CAS}$ | 22-22-22 |
| DRAM Refresh Timings: $T_{RFC}$ | 350ns |

We chose all benchmarks from the SPEC CPU2006 suite. We warm up the caches for 2 Billion instructions and execute 2 Billion instructions. To ensure adequate representation of different phases and regions, the 2 Billion instructions are collected by sampling 200 Million instructions per 1 Billion instructions over a 20 Billion instruction window. We execute all benchmarks in rate mode, in which all four cores execute the same benchmark. As shown in Table 2.2, we also create fifteen 8-threaded mixed workloads by forming three categories of SPEC2006 Benchmarks,

low MPKI, medium MPKI, and high MPKI by randomly picking benchmarks from each category to form mixed workloads. We perform timing simulation until all the benchmarks in the workload finish execution.

Table 2.2: Workload Mixes

| | |
|---|---|
| mix1 | gamess, tonto, perlbench, calculix, povray, hmmer, namd, h264 |
| mix2 | gromacs, sjeng, povray, calculix, astar, perlbench, hmmer, namd |
| mix3 | tonto, sjeng, gobmk, namd, povray, astar, calculix, perlbench |
| mix4 | astar, calculix, gamess, gromacs, povray, namd, h264, hmmer |
| mix5 | sjeng, gobmk, povray, hmmer, calculix, perlbench, gromacs, h264 |
| mix6 | sphinx, mcf, xalancbmk, bzip2, gcc, omnetpp, GemsFDTD, dealII |
| mix7 | milc, cactusADM, zeusmp, xalancbmk, omnetpp, soplex, dealII, GemsFDTD |
| mix8 | soplex, lbm, bwaves, omnetpp, sphinx, gcc, xalancbmk, cactusADM |
| mix9 | bwaves, cactusADM, omnetpp, milc, xalancbmk, wrf, leslie3d, mcf |
| mix10 | leslie3d, bzip2, libquantum, soplex, GemsFDTD, sphinx, zeusmp, lbm |
| mix11 | gamess, sphinx, mcf, tonto, xalancbmk, bzip2, perlbench, gcc |
| mix12 | omnetpp, GemsFDTD, bzip2, dealII, milc, cactusADM, zeusmp, calculix |
| mix13 | xalancbmk, milc, calculix, povray, omnetpp, soplex, hmmer, namd |
| mix14 | dealII, soplex, GemsFDTD, h264, gromacs, lbm, sjeng, povray |
| mix15 | bwaves, omnetpp, calculix, astar, sphinx, perlbench, gcc, povray |

## 2.5 RESULTS

In this section, we showcase the performance, power, and sensitivity results.

*Performance Impact on Plumber-R*

Figure 2.10 shows the speedup of Plumber-R when compared to a baseline system FR-FCFS thatis not secure against Drip-R vulnerability. On average, Plumber-R has a slowdown of only 15%. Close Page policy that prevents row-conflict side-channels has a slowdown of 36%. Our analysis shows that *mcf* and *libquantum* benefits the most Plumber-R performance optimization. Among low MPKI benchmarks, *povray* and *tonto* benefit the most from using Plumber-R. Furthermore, none of our benchmarks underperform as compared to close page policy. The memory intensive benchmarks reap most of the benefits from employing an open row-buffer.

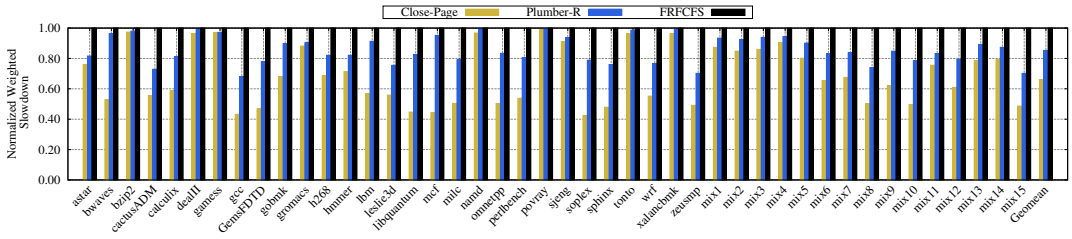


Figure 2.10: The performance of Plumber-R policy as compared to a baseline system that uses FR-FCFS scheduling policy. On average, Plumber-R as a slowdown of only 15% while providing mitigation against Drip-R vulnerabilities and providing row-buffer hit-rates. On the other hand, Close Page encounters a slowdown of 36%.

*Performance Impact on Plumber-Q*

Figure 2.11 shows the speedup of Plumber-R when compared to a baseline system FR-FCFS thatis not secure against Drip-Q vulnerability. On average, Plumber-Q has a slowdown of only 45%. The Fixed Service policy that prevents read contention side-channels has a slowdown of 60%. Our analysis shows that *lbm* and *libquantum* benefits the most Plumber-Q performance optimization. Among low MPKI benchmarks, *povray* and *dealII* benefit the most from using Plumber-Q. Furthermore, some of the benchmarks like mix14, outperform Plumber-Q. This is because, Plumber-Q may sometimes converge on inefficiently large or small numbers of time-epoch.

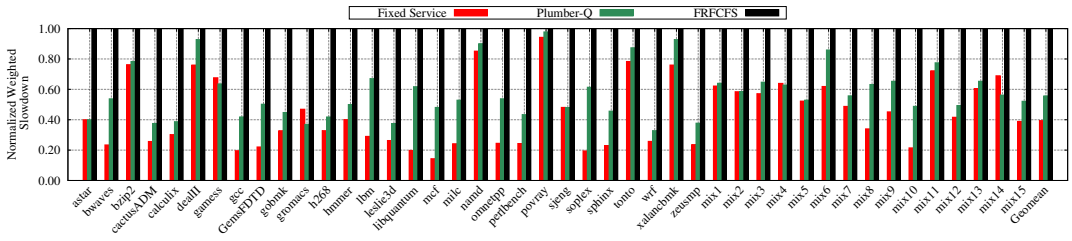


Figure 2.11: The performance of Plumber-Q policy as compared to a baseline system that uses FR-FCFS scheduling policy. On average, Plumber-R as a slowdown of only 45% while providing mitigation against Drip-Q vulnerabilities and providing row-buffer hit-rates. Fixed Service encounters a slowdown of 60% as it does not allow any row-buffer hit-rates.

*Power Consumption*

Figure 2.12 shows the normalized power consumption of memory systems that use Plumber-R and Plumber-Q as compared to a system that uses FR-FCFS. On average, Plumber-Q consumes the lowest amount of power as it does not issue fake requests as compared to Fixed Service. Overall, Plumber-Q consumes 50% lower power as compared to FR-FCFS. Furthermore, Close Page consumes 30% lower power. As Plumber-R performs well, it also expends power to service multiple reads and writes while keeping the row-buffer open. Therefore, Plumber-R consumes almost 90% of the power as compared to FR-FCFS.

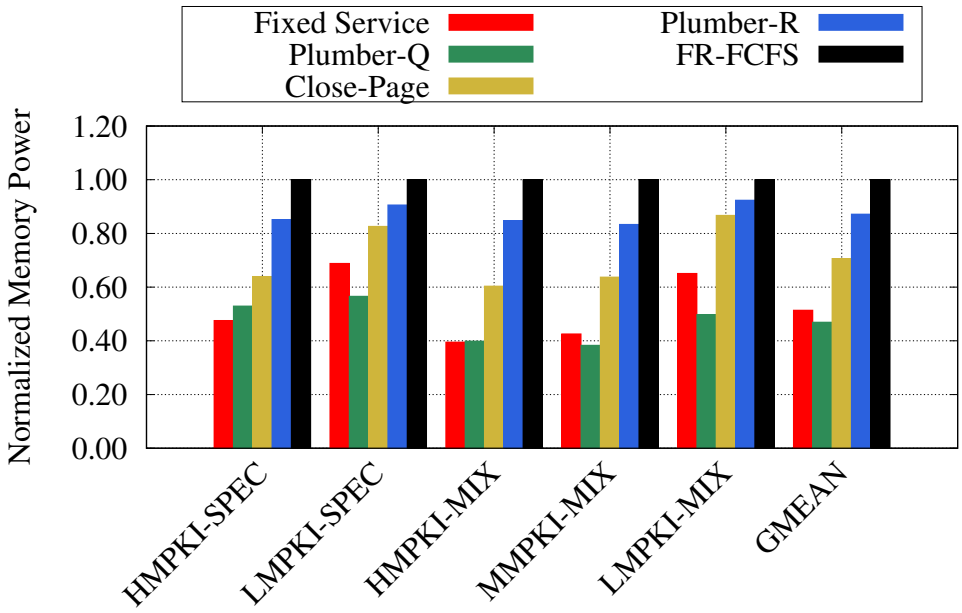


Figure 2.12: The effective memory power consumption. On average, Plumber-R and Plumber-Q consume 10% and 50% lower power as compared to FR-FCFS.

*Sensitivity to Channels*

Figure 2.13 shows the sensitivity to the number of channels with performance. As the number of channels increase, the performance of the machine increases. This is

because these greater number of channels expose bandwidth. This bandwidth can be wasted with much smaller performance penalties. For instance, Close Page and Plumber-R both show an increase in performance as bandwidth improves. However, Fixed Service and Plumber-Q show a reduction in performance. This is because, relatively, the FR-FCFS improves as channels increases. This causes the relative (normalized) performance of Fixed Service and Plumber-Q to start reducing.

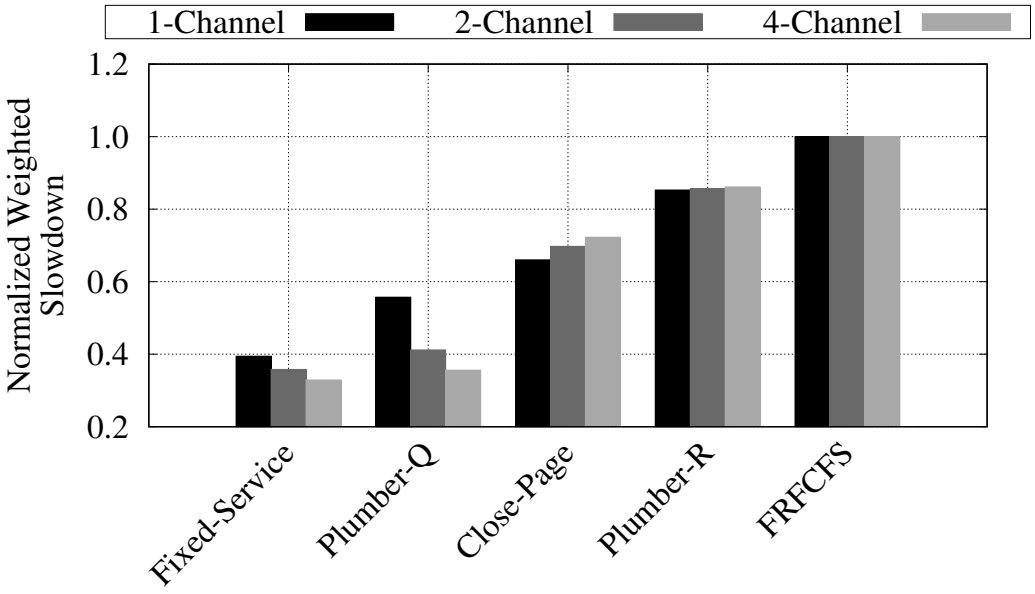


Figure 2.13: The performance of scheduling policies as the number of channels increases. On average, Close Page, Plumber-R, and FR-FCFS benefit from increasing bandwidth. However, Fixed Service and Plumber-Q reduce their relative performance as bandwidth increases. The performance of Fixed Service reduces the most, by nearly 75%.

*Sensitivity to the Number of Cores*

Figure 2.14 shows the sensitivity of different scheduling policies as the number of cores are increased. All policies benefit from having a lower number of cores as this reduces the pressure on the memory system. Furthermore, FR-FRCFS and Plumber-

R consistently perform well for 4 core and 8 core systems. The performance of Close Page, Plumber-Q and Fixed Service deteriorate as the number of cores increases. This is because, as Fixed Service and Plumber-Q try to reduce memory contention, an increased number of cores only increases the contention. Similarly, the Close Page policy ends up frequently closing most pages in 8 core system.

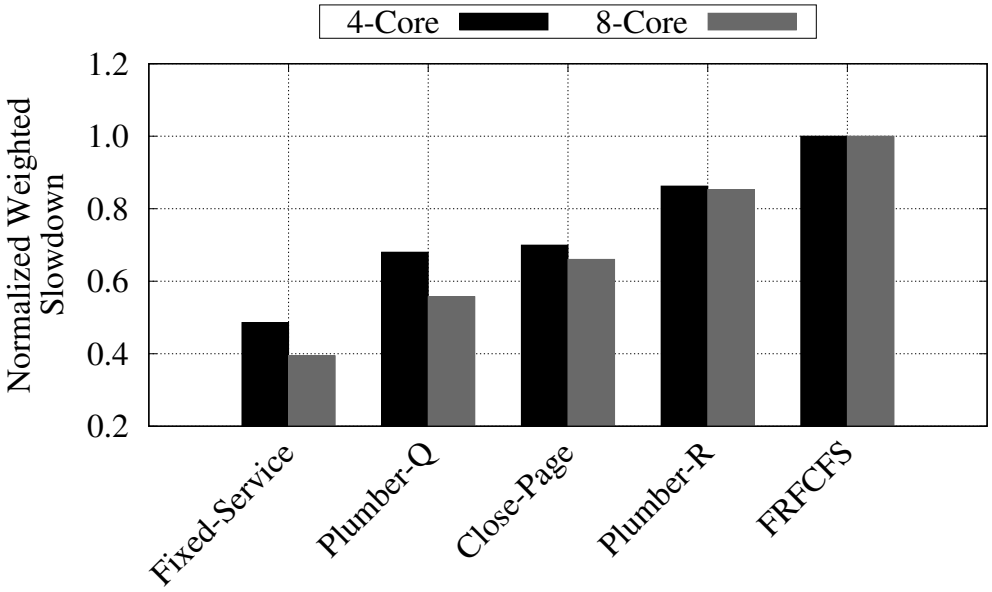


Figure 2.14: The effect of the number of cores on scheduling policies. As the number of cores increase, policies that help mitigate contention side channels like Fixed Service and Plumber-Q perform poorly. However, policies like Plumber-R and FR-FCFS that help maximize row-buffer hits perform well. The performance of Fixed Service policy reduces by the largest margin of 60%.

## 2.6 RELATED WORK

We highlight some of the prior work in the area of memory security, covert channels and side-channels

*Row Buffer Conflict Vulnerabilities*

Some prior works have proposed solutions to mitigate this vulnerability. For instance, Wang et al. design a memory scheduler with temporal partitioning [22]. This scheduler introduces dead time slots between requests. Furthermore, to increase isolation, the temporal partitioning scheduler implements a per security domain based queuing structure within the memory controller. In similar vein, prior work have proposed bank triple alternation (BTA) to utilize bank-level parallelism. BTA ensures that two memory requests to the same bank from different consecutive domains need to delay and wait for a predetermined worst-case period. This helps the BTA scheduler remove dead-time as it can keep all its banks busy [17]. Unlike these prior work, our Plumber-R policy does not require security domains. Furthermore, to utilize row-buffer hit-rates, we also introduced an epoch-based. When compared to prior work, our approaches in Plumber-R help provide dramatic performance improvement.

*Read Queue Contention Vulnerabilities*

Similar to row-conflicts, our work also shows that read queue contentions can create side/covert channels. Prior work like fixed-service [17] have solved this by shaping the memory access pattern, resulting in a high performance cost. As we can see from Figure 2.1(b), the read queuing delay of Fixed Service scheduler is much longer than a "not secure" FR-FCFS scheduling policy. As a contrast to these prior work, in our Plumber-Q policy, we reduced read queuing delay by using epochs across processes and taking advantage of row-buffer hits. We show that the concept of epochs can be broadly applied for row-conflict and read queue contention mitigation.

*General Timing Channel Mitigation*

General timing channels mitigation across related disciplines (such as caches and other shared resources) have also been examined in prior work. Martin et. al suggest to restrict the user's ability to take fine-grained timing measurements (such as `rtdscp`) [23]). The detection of timing channels via their architectural interference has also been proposed [24]. We believe that some of our ideas in epoch creation and locality can be re-used to improve the performance of caches while also providing security. Several authors have proposed using attacks like Flush-Reload that aggressively trash the cache and evict blocks [25]. These prior work use these evicted block information to relay information

## 2.7 SUMMARY

As machine resources are being share for efficiency, they also present avenues for side-channels and covert-channels. Today, several researchers in Academia and Industry are tacking side-channels in caches. This thesis proposes Drip-R and Drip-Q which target the main memory system. Thereafter, this paper proposes two high-performance scheduling policies called Plumber-R and Plumber-Q to mitigate these vulnerabilities. Overall this paper has the following contributions:

- This thesis suggests strategies that help launch these vulnerabilities without having prior knowledge of the address mapping of the target machine.
- We propose two high-performance scheduler called Plumber-R and Plumber-Q, that mitigate Drip-R and Drip-Q vulnerabilities.
- To improve performance, we use the concept of epochs in time and queues. Using epochs, we try to schedule requests to the same row-buffer and improve the row-buffer hit-rate. This helps improve performance.
- Plumber-R and Plumber-Q schedulers provide 29% and 45% higher speedup

than prior state-of-the-art scheduling policies like Fixed Service and Close Page.

- Furthermore, the Plumber-R and Plumber-Q schedulers also reduce memory power consumption by 10% and 50% respectively when compared to a FR-FCFS scheduler.

As systems with large memory capacities and new memory technologies are being developed, we believe that this thesis provides key design choices to improve their security while maintaining performance.

# Bibliography

[1] T. Zhang, Y. Zhang, and R. B. Lee, "Dos attacks on your memory in cloud," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3052973.3052978

[2] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. Berkeley, CA, USA: USENIX Association, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3241189.3241209

[3] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware trojans through side-channel engineering," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '09. Berlin, Heidelberg: Springer-Verlag, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04138-9_27

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. Berkeley, CA, USA: USENIX Association, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277276

[6] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. Berkeley, CA, USA: USENIX Association, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277277

[7] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018.

[8] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019.

[9] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358314

[10] "Engineering New Protections Into Hardware," https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html, accessed: 2019-11-23.

[11] "AMD Product Security," https://www.amd.com/en/corporate/product-security, accessed: 2019-11-23.

[12] J.-S. Kim, C. S. Oh, H. Lee, D. Lee, H.-R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S.-K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, K. Kyung, J.-S. Choi, and Y.-H. Jun, "A 1.2v 12.8gb/s 2gb mobile wide-i/o dram with 4x128 i/os using tsv-based stacking," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, Feb 2011.

[13] JEDEC Standard, "DDR3 Standard," in *JESD79-3E*, 2015.

[14] *ddr3_8gb_1.5v_twindie_x4x8.pdf - Rev. C 4/13 EN*, Micron, 2011.

[15] K.-N. Lim, W.-J. Jang, H.-S. Won, K.-Y. Lee, H. Kim, D.-W. Kim, M.-H. Cho, S.-L. Kim, J.-H. Kang, K.-W. Park, and B.-T. Jeong, "A 1.2v 23nm 6f2 4gb ddr3 sdram with local-bitline sense amplifier, hybrid lio sense amplifier and dummy-less array architecture," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, Feb 2012.

[16] JEDEC Standard, "DDR4 Standard," in *JESD79-4*, 2015.

[17] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari, "Avoiding information leakage in the memory controller with fixed service policies," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015.

[18] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "{DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[19] 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*, ser. ISCA '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/339647.339668

[20] M. Blackmore, "A quantitative analysis of memory controller page policies," 2013.

[21] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module a simulation infrastructure for the jwac memory scheduling championship," 2012.

[22] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Dec 2006.

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

[24] C. Hunger, M. Kazdagli, A. Rawat, A. 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)*, Feb 2015.

[25] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom