# THE PERFORMANCE COST OF SECURITY

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Lucy Bowen
June 2019

COMMITTEE MEMBERSHIP

TITLE: The Performance Cost of Security

AUTHOR: Lucy Bowen

DATE SUBMITTED: June 2019

COMMITTEE CHAIR: Christopher Lupo, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Maria Pantoja, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Bruce DeBruhl, Ph.D.
Professor of Computer Science

ABSTRACT

The Performance Cost of Security

Lucy Bowen

Historically, performance has been the most important feature when optimizing computer hardware. Modern processors are so highly optimized that every cycle of computation time matters. However, this practice of optimizing for performance at all costs has been called into question by new microarchitectural attacks, e.g. Meltdown and Spectre. Microarchitectural attacks exploit the effects of microarchitectural components or optimizations in order to leak data to an attacker. These attacks have caused processor manufacturers to introduce performance impacting mitigations in both software and silicon.

To investigate the performance impact of the various mitigations, a test suite of forty-seven different tests was created. This suite was run on a series of virtual machines that tested both Ubuntu 16 and Ubuntu 18. These tests investigated the performance change across version updates and the performance impact of CPU core number vs. default microarchitectural mitigations. The testing proved that the performance impact of the microarchitectural mitigations is non-trivial, as the percent difference in performance can be as high as 200%.

ACKNOWLEDGMENTS

My thanks to:

My husband Chad, for supporting me through the trials and tribulations of both my bachelors and masters degrees. Doing this alone would have been far more difficult, and I would not have flown as high.

My parents, Nikki and Tommy, for their support, as well as giving me the desire to go into this field in the first place.

Dr. Lupo, both for being a great advisor on this project, and for walking back the feature creep I attempted to introduce every meeting.

Mr. Scovil, whose initial classes and teaching gave me the foundation to get this far.

All of my friends, who either helped me study throughout college or just put up with my whining.

Lastly to Andrew Guenther, for this LaTeX template that saved a huge amount of time on formatting.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

# INTRODUCTION

Performance has always been one of the most important factors in computer design [36]. As figure 1.1 shows, over time the gap in performance between the memory and CPU has widened. In order to take advantage of the CPU speed, various microarchitectural optimizations were created. Sadly, recent research shows these optimizations can leak privileged information if exploited by savvy attackers [19]. Processor manufacturers and operating system programmers have created software based solutions to these exploits, they are not all-encompassing, and some exploits can only be solved with hardware changes [44]. Several software solutions are harmful to performance, but these should be temporary measures as new hardware is designed.

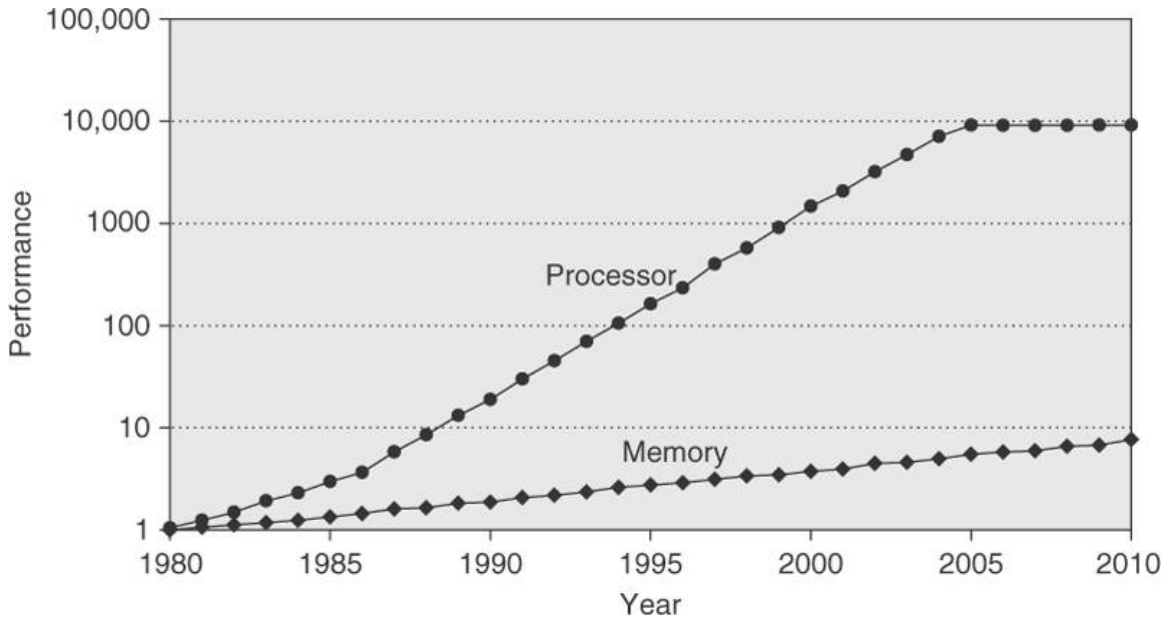



**Figure 1.1: Performance: Memory vs. Process [36]**

It is important to quantify how much performance has been lost for the sake of security. Users are allowed to disable these mitigations, even if it is to their own detriment. Those with performance-intensive tasks: simulation running, image creation / rendering, video encoding, etc., may desire to regain the previous performance. A user could chose to take the risk with removing the mitigations, or if they wanted to keep some security, instead use a dedicated machine for those tasks and ensure that it contains no sensitive information.

This paper is a comprehensive analysis of the performance impact from the software based mitigations on different patch levels of Ubuntu 16 and Ubuntu 18, and the affect of allocating differing numbers of cores to the latest patch. By using a test-suite consisting of forty-six different tests, a quantitative summary of the performance impacts was created. Analysis of these tests showed that the mitigations had a quantifiable affect, with some being negligible but others has as much as three orders of magnitude difference in performance.

Chapter 2 provides background into the various microarchitectural optimizations and the attacks on them. The design and implementation of experiment are covered in Chapters 3 and 4, with an explanation of the tests in the test-suite in the former and an explanation of the software mitigations in the latter. Chapter 5 presents an analysis of both the individual test results and of the results when compared as a group. Potential future work and conclusions are presented in Chapters 6 and 7, respectively.

Chapter 2

# BACKGROUND

Many of the microarchitectural optimizations that exist within modern processors were first created decades ago. A sampling of these optimizations can be seen in table 2.1, along with their year of introduction to consumer hardware. However, these same optimizations that are critical to modern design and performance, have recently been proven to have massive, difficult to fix, security flaws. A previously dangerous security attack type, cache timing attacks, have been adapted to attack how these optimizations work together to increase performance. By doing this, attackers can leak sensitive data without a trace.

This chapter covers how different microarchitectural optimizations work, how cache timing attacks work, and what the recent high profile microarchitectural attacks are.

**Table 2.1: Optimization and Year Implemented by Intel**

| Optimization | Intel Micro-Architecture | Year |
|---|---|---|
| Instruction Pipelining | 8086 | 1978 |
| Dynamic Execution | i686 | 1995 |
| Simultaneous Multithreading | NetBurst | 2002 |
| Cache L1 | i486 | 1989 |
| Cache L2 | i686 | 1995 |
| Cache L3 | Nehalem | 2008 |

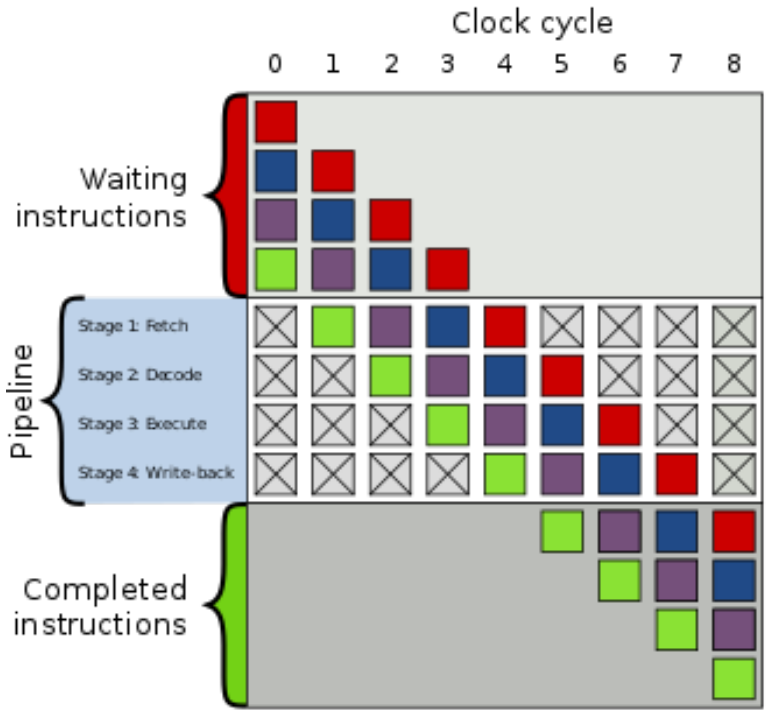



**Figure 2.1: Pipelining Diagram [23]**

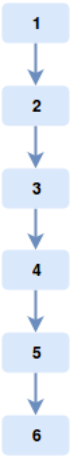
## 2.1 Instruction Pipelining

Instruction Pipelining is a technique where multiple instructions are overlapped in execution in order to increase throughput. As seen in Figure 2.1, the pipeline is divided in stages where each stage completes part of an instruction in parallel. The time required to move an instruction one step further in the pipeline is a machine cycle. Due to how the stages interact, all stages must be ready to proceed at the same time, and the length of the machine cycle is determined by the time required for the slowest pipe stage. Because of this, modern pipelines have been increasing in depth by splitting up the pipeline stages into increasingly smaller chunks [36].

| Instruction Number | Instruction Type | Target Register | Memory Location 1 | Memory Location 2 | Latency Cycles |
|---|---|---|---|---|---|
| 1 | LD | F2 | cache | | 4 |
| 2 | LD | F4 | mem | | ~ 90 |
| 3 | MULTD | F6 | F4 | F2 | 4 |
| 4 | SUBD | F8 | F2 | F2 | 6 |
| 5 | DIVD | F12 | F2 | F8 | 103 |
| 6 | ADDD | F10 | F6 | F12 | 6 |
| | | | | Total: | 213 |

(a) Assembly Instructions

In Order Execution




(b) In Order

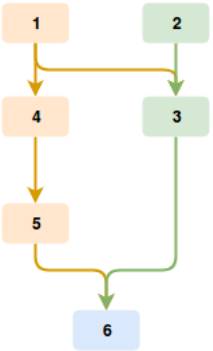
Execution Flow

**Figure 2.2: In Order Execution**

| Instruction Number | Instruction Type | Target Register | Memory Location 1 | Memory Location 2 | Latency Cycles |
|---|---|---|---|---|---|
| 1 | LD | F2 | cache | | 4 |
| 2 | LD | F4 | mem | | ~ 90 |
| 3 | MULTD | F6 | F4 | F2 | 4 |
| 4 | SUBD | F8 | F2 | F2 | 6 |
| 5 | DIVD | F12 | F2 | F8 | 103 |
| 6 | ADDD | F10 | F6 | F12 | 6 |
| | | | | Total: | 119 |

(a) Assembly Instructions

Out of Order Execution




(b) Out of Order

Execution Flow

**Figure 2.3: Out of Order Execution**

## 2.2 Dynamic Execution

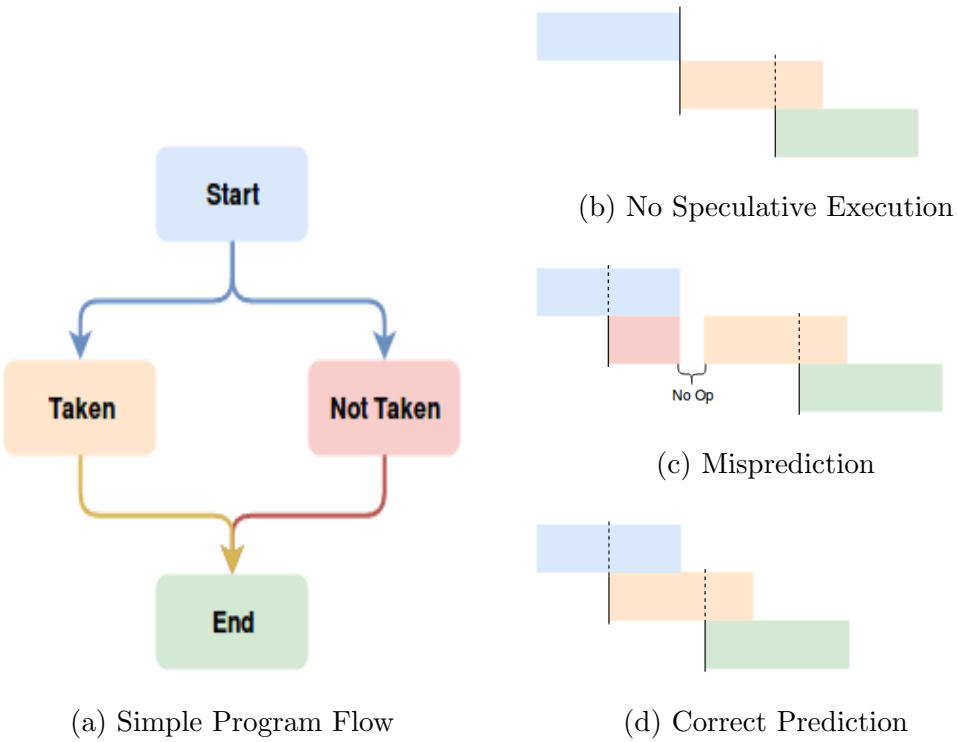
Dynamic execution, also called out-of-order execution, is a collection of optimizations designed to utilize potentially wasted clock cycles. They allow a processor to avoid stalls when the data needed to perform an operation is unavailable. Comparing Figure 2.2 to Figure 2.3 shows the performance gained when switching to out-of-order execution; information from [1, 31]. Dynamic execution has a greater benefit with longer pipelines, and when the clockspeed of the CPU is significantly higher than the speed of the main memory bank [36]. Dynamic execution is composed of three parts: data flow analysis, speculative execution, and branch prediction.

### 2.2.1 Data Flow Analysis

Data-flow analysis is used to align instructions for optimal execution, as opposed to executing them in the order they came in. This allows for a CPU to fill empty pipeline slots with instructions that are ready to execute. At the end of the process the results are re-ordered so that it appears that instructions were processed normally. The sequential ordering of the original code is program order, while the changed processor order is data order [36]. Complex circuitry is needed to convert from one ordering to the other while maintaining logical ordering.

### 2.2.2 Speculative Execution

Speculative execution is a CPU optimization where a processor executes instructions that may be incorrect. This allows the CPU to work ahead when it has idle time, i.e. when there is a conditional branch that has an as of yet, unknown direction. The CPU uses branch prediction to guess the direction, and save the current register state as




(a) Simple Program Flow

(b) No Speculative Execution

(c) Misprediction

(d) Correct Prediction

**Figure 2.4: Speculative Execution Pipelines**

before subsequent instructions start executing. Afterwards, when the direction of the branch is known, the guess made is validated. If the guess was incorrect, the program state is reverted back and the execution of the correct path is initiated instead, while the pending instructions from the incorrect guess are abandoned [19]. The potential performance gain from this working ahead can be seen in Figure 2.4. Figure 2.4 also shows that the performance impact when incorrect is negligible.

### 2.2.3 Branch Prediction

Branch prediction is an optimization that allows for a CPU to accurately guess the direction of conditional branches in order to start speculative execution. When a branch instruction is encountered, the processor must make a prediction on the direction it will take, either "taken" or "not taken". A conditional branch that is "taken"

is a jump that goes to a different location in program memory where the second branch of code is stored, while a "not taken" branch falls through the block to the code which immediately follows the jump. The processor does not know for certain if a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline. In order to make an accurate guess, a predictor will have both local and global predictors [36]. To further increase performance, whenever a branch instruction is executed, the corresponding correct jump location is cached in the Branch Target Buffer (BTB) where it can be used as a guess for the same branch instruction when it is executed again later [19].

## 2.3 Simultaneous Multithreading

Simultaneous multithreading is an optimization that allows for multiple independent threads of execution to use a shared CPU core. This optimization is how the CPU can have not only multiple threads executing simultaneously, but also multiple processes which have their own page tables, privilege levels, I/O permissions, etc. The threads and processes running on the same core are considered completely separated [36].

Simultaneous multithreading allows for instructions from more than one thread to be executed in any given pipeline stage at a time. The number of concurrent threads can be as few as two concurrent threads per CPU core, but some processors support up to eight. Extra threads can also be used proactively to seed a shared resource like a cache, to improve the performance of another single thread, or to provide redundant computation for error detection and recovery [27].

## 2.4 CPU Caches

The CPU cache is a small, fast memory module located close to the processor core. By buffering frequently used data, it is able to hide the slow latency to main memory. Modern CPU architectures implement n-way set-associative caches, where the cache is divided into cache sets, and each cache set comprises n cache lines. A line is loaded in a set depending on its address, and each line can occupy any of the n ways. The processors cache replacement policy dictates the cache line that is replaced when new data is loaded, so that the CPU can achieve optimal cache usage [36]. If a program requests data that is already in the cache, that is a cache hit. However, if the data requested is not in the cache, and needs to be fetched from a lower level - that being a lower cache level, main memory, or the hard-drive - it is called a miss. The unprivileged `clflush` instruction evicts a cache line from the entire cache hierarchy, but a program can also evict a cache line by accessing enough memory. As seen in 2.5, modern Intel processors have at least three cache levels, with one or more of those levels shared between the cores on multi-core processors [66].

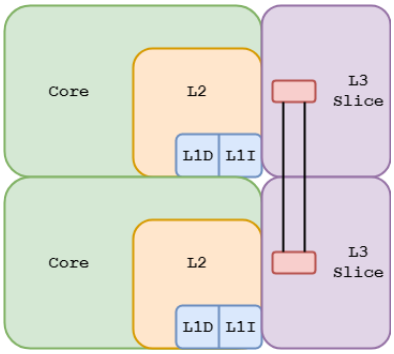



**Figure 2.5: Cache Levels on Multicore Architecture**

### 2.4.1 L1 Cache

The L1 cache is also known as the "primary cache", and is the highest level of cache. This is because it is the closest to the processor and the fastest memory in the computer. Figure 2.5 shows that it is both the smallest cache level and is divided into two distinct caches, one for memory and one for instructions [36, 66]. The instruction cache works as an input cache, and is particularly efficient when the program starts to repeat a small part of itself, i.e. a loop. In some micro-architectures the L1 instruction cache is used to store additional control-flow data, in order to speed up the decoding process. The data cache is both an input and an output cache; it is used to store the data that is going to be written back to memory as well as the data that was most recently used [36].

### 2.4.2 L2 Cache

The L2 cache feeds the L1 cache, and is made of memory that is slower than L1 memory. Before being placed on chip in 1995, it existed on the motherboard, and was therefore impacted by bus speed [66]. Unlike the L1 cache, which is always unique to a core, the L2 cache varies depending on the micro-architectural design. It may be shared between all cores, groups of cores (i.e. each set of two or four), or as figure 2.5 shows, be individual to each core [40]. Most Intel L2 caches are inclusive of L1, i.e. all data within the L1 data and instruction caches are also stored within it. This allows for data that is evicted from L1 to be retrieved quickly from L2; this is useful when the working set of a program is larger than the size of the L1 cache. For this to work, L2 associativity must be equal to or greater than L1 associativity irrespective of the number of sets [36].

### 2.4.3 L3 Cache

The L3 cache, also called last-level cache, is often shared between all CPU cores, and is inclusive of all cache levels above it, i.e. the data within all L1 and L2 caches is also present in the L3 cache. Newer Intel processors, Sandy-Bridge (2011) and later, divide the L3 into per-core slices, which are connected by a ring bus; figure 2.5. The ring bus is a scalable bus that allows many physical cores to connect to the cache at once, while allowing for the cache to run at core clock speed [61]. Slices can be accessed concurrently and are effectively separate caches, although the bus ensures that each core can access the full L3, albeit with higher latency for remote slices [48]. Because of the inclusively, executing code or accessing data on one core has immediate consequences even for the private caches of the other cores [66]. This can be exploited in cache attacks.

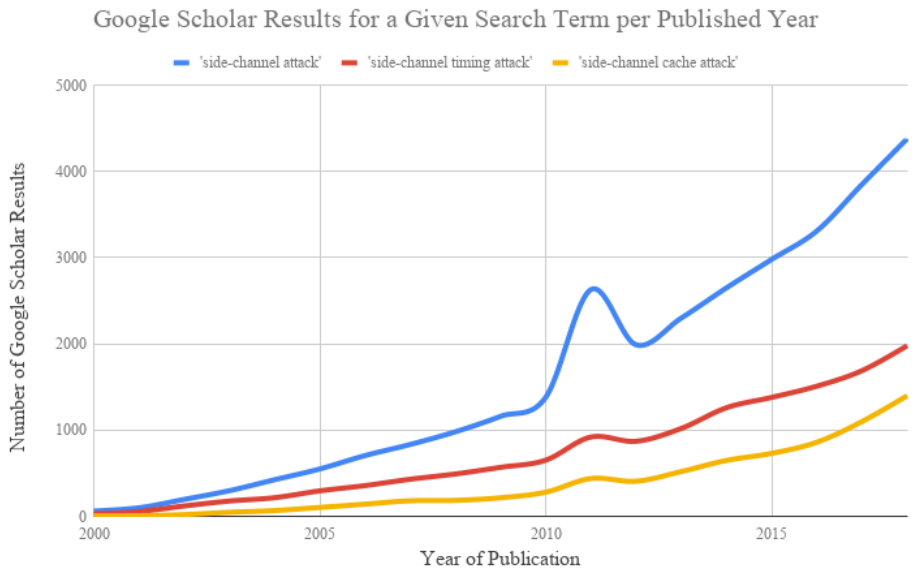



**Figure 2.6: Increasing Prevalence of Side-Channel Attacks in Literature**

## 2.5 Cache Attack

Historically in computer security, solutions which used powerful encryption/decryption algorithms with cryptographic keys were considered secure. If a cryptographic algorithm is given a large enough key, brute force attacks become computationally infeasible. Therefore, sometimes attackers instead target the physical implementation in hardware in order to take advantage of some physical information leaked by a cryptographic device. This is a side channel attack, where the information is gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself [19]. Figure 2.6 shows that this type of attack and its sub-types are becoming more common [65].

A timing attack is an attack based on measuring how much time various computations take to perform. Cache attacks are a subset of timing attacks, which focus on exploiting the timing differences caused by the lower latency of CPU caches when compared to physical memory [19]. This attack targets the micro-architectural design of the CPU by monitoring the cache accesses in a shared physical system and is commonly used against virtualized environments or cloud services [75]. These attacks have a similar execution pattern where the attacker finds a way to manipulate the cache to a known state, waits for victim activity and then examines what has changed. While other attacks exist, the three most well known are the Prime + Probe attack, the Evict + Time attack, and the Flush + Reload attack, with many of the other existing cache attacks being derivations or combinations of them.

### 2.5.1 Prime + Probe

The Prime + Probe attack was one of the first two cache attacks discovered, in 2005. It is performed by priming the victim's cache by filling it with known attacker addresses. While the victim carries out normal tasks, some attack data will be evicted from the cache. When the attacker probes the primed space, they can time the access time to learn if memory congruent with a cache set was used. By doing so, the attacker gains intimate knowledge of both the victims activities, and the contents the victim replaces controlled data with [52]. This attack can target both static and dynamically allocated memory, but only works with inclusive caches. While the attack does not need to share memory with the victim, it does need to share the same CPU socket. It requires live analysis of the data being replaced, and is only accurate to cache set congruence, but is performable using Javascript [19].

### 2.5.2 Evict + Time

The Evict + Time attack was the other cache attack discovered in 2005. To perform it, the attack must execute and time a function that primes the cache. Then, the attacker must evict a line from the cache, and re-time the priming function. If the function is faster the second time it is called, it likely used an address congruent to the cache set [52]. In its most basic form this attack is only accurate to cache set congruence. While it requires a function call to be executed, the results can be analyzed later instead of live, and the attack is possible in Javascript [19].

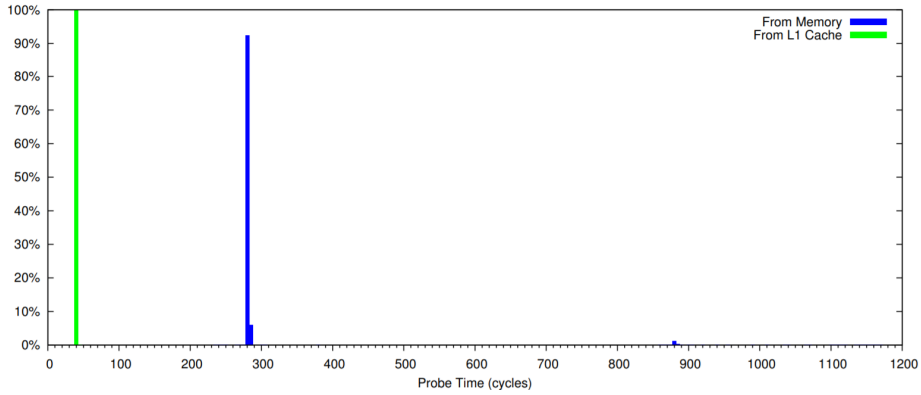



**Figure 2.7: Distribution of Load Times [75]**

### 2.5.3 Flush + Reload

Flush + Reload was discovered in 2014 and is a generic last level cache side-channel attack that takes advantage of executable code page sharing between processes. When a user runs a program, the operating system loads the program into physical memory. If additional users run the same program, rather than loading duplicate tables, the operating system will instead set each additional user's page tables to use a copy of the page table that was loaded into memory for the first user. This physical memory is read-only, and thus there should be no difference between one to n users reading it. However, if one user is instead an attacker using Flush + Reload, they can watch a number of cache lines, called "probes," which are specified as addresses in the victim program's executable code. By flushing these lines out of the cache, waiting, then timing how long it takes to read them, the attacker can determine if they were replaced in the cache – a fast read after the flush means the victim replaced them by accessing them, while a slow read means the victim hasn't [75]. Figure 2.7 shows the dramatically different access times when accessing lines that were replaced. This attack is the fastest cache attack, and with cache line accuracy, the attack with the best resolution. However, it requires shared memory with the victim, does not work on Javascript, and requires a live analysis of the data [19].

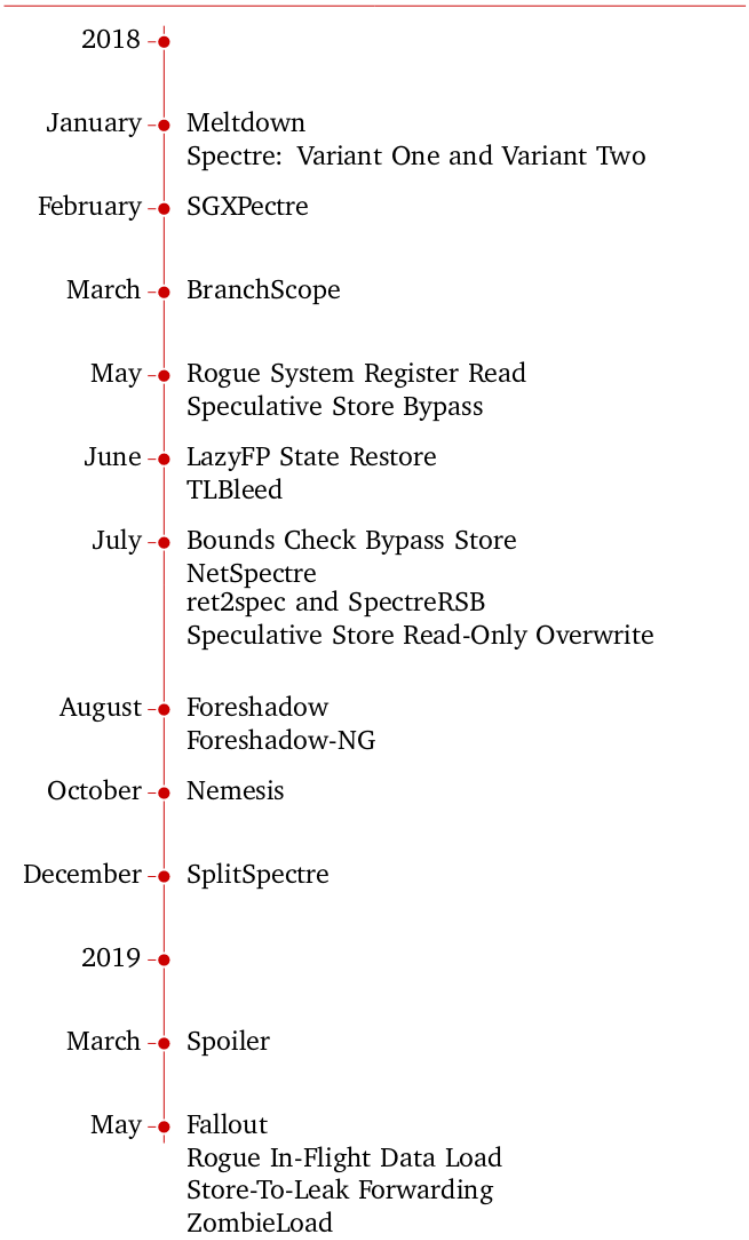



**Figure 2.8: Timeline of Microarchitectural Attacks**

## 2.6 Microarchitectural Attacks

Microarchitectural attacks are attacks which exploit the effects of microarchitectural components or optimizations. Since January 2018 the field has been in the news due to high profile, difficult-to-fix attacks, specifically ones based on the Meltdown and Spectre attacks. Figure 2.8 is a timeline of published research since the initial release of Meltdown and Spectre and it shows that even with an embargo of up to a year, research in this field is booming. Meltdown- and Spectre-type attacks are both speculative execution based attacks; the former attacks obtain data from instructions following a fault, while the latter attack prediction units [19]. Microarchitectural Data Sampling is a new attack type that instead targets small frequently overwritten buffers used in speculative execution. The microarchitectural attacks Nemesis, TLBleed and Spoiler use concepts from the above attack types but are not, themselves, members of them.

### 2.6.1 Meltdown-Type Attacks

Meltdown, once a single attack, has become the name of an overarching attack type where information is gained from transient executions following a faulting instruction. This attack type exploits out-of-order execution by extracting data from faulting instructions that are forwarded ahead in the pipeline. Meltdown attacks use transient execution to "melt down" architectural isolation barriers by computing on unauthorized results of faulting instructions, thus transiently bypassing hardware-enforced security policies in order to leak data that was architecturally inaccessible for the application. They reflect a failure of the CPU to respect hardware-level protection boundaries for transient instructions [22].

**Figure 2.9: Meltdown Attack Logo [47]**

#### 2.6.1.1 Original Meltdown

Meltdown is a security vulnerability that exploits a race condition between memory access and privilege checking during instruction processing. When running code speculatively, there are no permission checks to see if the memory accesses from cache are accessing privileged memory. Instead, instructions are run as fast as possible in order to not waste CPU resources. During this window without permission checks the process can access permission protected memory and write to the cache. The permission check is only performed after the decision about committing to the speculation. However, because the cache is modified regardless of if the speculation is committed or not, an attacker can use this to modify the cache. Later they can access those addresses legitimately, gaining access to data previously inaccessible to it [47].

**Figure 2.10: Foreshadow Attack Logo [70]**

#### 2.6.1.2 Foreshadow

Foreshadow is a Meltdown variant which exploits Intel's Software Guard Extensions. Intel designed the Software Guard Extensions (SGX) to allow user-level code to allocate private regions of memory, called enclaves, that are protected from processes running at higher privilege levels. Normally, when there is an attempt to read memory, a process can use speculative execution to modify the cache based on the data that was read. When the processor detects that this process involves enclave memory instead of normal memory, and that reading is not permitted, then the processor is allowed to block speculation. Because speculation has been blocked, there should be no cache modification. However, if the read is from Level 1 cache, speculative execution can finish before the blocking permission check has finished [70].

#### 2.6.1.3 Foreshadow-NG

Foreshadow-NG is a continuation on Foreshadow which proved that the attack was not limited to Intel SGX and could instead be used for extracting any information residing in the L1 cache. This attack completely bypasses the virtual memory abstraction by directly exposing cached physical memory contents to unprivileged applications and guest virtual machines. An attacker could get information belonging to anyone, including the System Management Mode, the Kernel, or other Virtual Machines running on third-party clouds [74].

#### 2.6.1.4 Rogue System Register Read

Rogue System Register Read is a Meltdown variant which exploits speculative system register reads. Processors can speculatively read system registers not accessible at

the current exception level, provided that it is a register that can be read without side-effects. This access returns a speculative register value which is used in subsequent speculative load instructions. If the speculation is not correct, the results are discarded, however, the data returned from the inaccessible system register can be used to perform further speculation. This speculation can be exploited by the speculation-based cache timing side-channels [38].

#### 2.6.1.5 Lazy FP State Restore

Lazy FP State Restore is a Meltdown-type exploit that targets the optimization called lazy FPU context switching. Lazy FPU context switching is an optimization where the operating system can defer the context switch of the FPU and SIMD register set until the first instruction is executed that needs access to these registers. This allows old content to be left in place in case the current task doesn't use those registers. The processor does this by offering the ability to toggle the availability of instructions utilizing floating point and SIMD registers, and if the instructions are turned off, any attempt of executing them will generate a fault, Using Lazy FP State Restore, an attacker can recover the FPU and SIMD register set of arbitrary processes or VMs by reconstructing the FPU and SIMD register content of other processes or virtual machines after the fault indicating the first use of FPU or SIMD instructions [63].

### 2.6.2 Spectre Type Attacks

Like Meltdown, the original Spectre attack, made up of Variants 1 and 2, has grown to encompass its own attack type. Spectre-type attacks exploit transient execution following control or data flow misprediction, and thus rely on dedicated control or dataflow prediction machinery. In Spectre-type attacks, transient instructions only

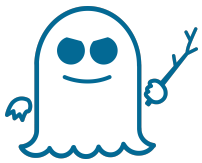


**Figure 2.11: Spectre Attack Logo [44]**

compute on data which the application is also allowed to access architecturally, which allows the attacks to transiently bypass software-defined security policies in order to leak secrets out of the program's intended code/data paths. Successful Spectre attacks steer a victim into transiently computing on memory locations the victim is authorized to access but the attacker is not, and then leaking the data. This leakage is an unintended side-effect of important performance optimizations, so mitigating Spectre-type attacks requires careful hardware-software co-design [22].

#### 2.6.2.1 Variant One

Variant One attacks the CPU branch predictor by mistraining it into mispredicting the direction of a branch. This causes the CPU to temporarily violate the program semantics by executing code that would not have been executed otherwise. This incorrect speculative execution allows an attacker to read secret information stored in the program's address space [44].

#### 2.6.2.2 Variant Two

Variant Two attacks the Branch Target Buffer, BTB, using techniques from return-oriented programming [60]. In return-oriented programming, the attacker gains control of the call stack to hijack program control flow and proceeds to execute specific machine instruction sequences already present in the machine's memory. These machine instructions are known as gadgets, and typically end in a return instruction.

They are normally located in a subroutine within the existing program and/or shared library code. Using Variant Two, the attacker chooses a gadget from the victim's address space and influences the victim to speculatively execute the gadget, but unlike previous return-oriented programming attacks, there does not need to be a vulnerability in the victim code. Instead, the attacker trains BTB to mispredict a branch from an indirect branch instruction to the address of the gadget, resulting in speculative execution of the gadget. While the effects of incorrect speculative execution are eventually reverted, their effects on the cache are not, thereby allowing the gadget to leak sensitive information via a cache side channel [44].

#### 2.6.2.3 SGXPectre

SGXPectre is a Spectre-type attack that uses branch prediction to leak secrets from SGX enclaves. The branch prediction of enclave code can be influenced by non-enclave programs, thus the control flow of the enclave program can be temporarily altered to execute instructions that lead to observable cache-state changes. Attackers can watch the cache state in order to learn the secrets inside enclave memory or its internal registers, thus defeating the SGX confidentiality guarantee [24].

#### 2.6.2.4 BranchScope

BranchScope is a Spectre-type attack that targets the directional component of the branch predictor. The attacking process infers the direction of an arbitrary conditional branch instruction in a victim program by manipulating the shared directional branch predictor. When forced, predictors will revert to the backup two bit predictor, allowing for attacks on complex hybrid predictors. By causing collisions between its branches and the branches of the victim process in the Pattern History Table (PHT)

an attacker can uncover the direction of the victim's branches, and is thus able to obtain secret bits with a low degree of error, even when the victim process was inside an Intel SGX [29].

#### 2.6.2.5 Speculative Store Bypass

Speculative Store Bypass is a Spectre-type attack that exploits speculative execution around memory stores. Processors can allow loads to speculatively execute even if the address of a preceding potentially overlapping store is unknown. This can potentially allow these loads to read stale data values speculatively. Although the processor will correct for such cases later, in the interim an attacker can use speculative execution to reveal the value of memory that is not normally accessible to them. This has been proven with proof of concept code using Flush + Reload and Prime + Probe, but only in managed runtimes, where an attacker is able to influence the generation of code [38].

#### 2.6.2.6 Bounds Check Bypass Store

Bounds Check Bypass Store is based off of Spectre Variant 1 and leverages speculative stores to create speculative buffer overflows, allowing it to modify data and code pointers. During speculative execution, the processor may ignore the bounds checks which provides an attacker with the full power of an arbitrary write. While this is only a speculative write, it can still lead to information disclosure. Using this, a proof of concept attack was created that used return-oriented programming techniques [43].

#### 2.6.2.7 Speculative Store Read-Only Overwrite

Speculative Store Read-only Overwrite is based off of both Spectre Variant 1 and Meltdown. By exploiting the lazy enforcement of the User/Supervisor protection flags for page-table entries, it can use the same mechanism as Meltdown to bypass the Read/Write page table entry flags. By doing this, speculative stores can overwrite read-only data, code pointers, and code metadata, including v-tables, GOT/IAT, and control-flow mitigation metadata. As a result, any sandboxing that depends on hardware enforcement of read-only memory is rendered ineffective. Additionally, the mitigations created for Meltdown do not affect this Spectre variant [43].

#### 2.6.2.8 SpectreRSB and ret2spec

SpectreRSB and ret2spec are Spectre-type exploits that target the Return Stack Buffer, RSB, in order to expose sensitive information. The two expoits were published separately but attack the same target with similar strategies. The RSB is a processor structure used to predict return address by pushing the return address from a call instruction on an internal hardware stack. When the return is encountered, the processor uses the top of the RSB to predict the return address to support speculation with very high accuracy. The SpectreRSB exploit shows that the RSB can be easily manipulated by user code by using a call instruction to cause a value to be pushed to the RSB. Then the stack can be manipulated by the user so that the return address no longer matches the RSB, thus allowing for an attack similar to Spectre v1. Using this exploit, proof of concept attacks have been shown to be successful on both local and SGX protected memory [45]. The other attack, ret2spec, also poisoned the RSB in order to force a process to execute arbitrary code speculatively, and thus report potential secrets. However, ret2spec has a more advanced, second attack, which

allows attackers to abuse RSBs to trigger speculation of arbitrary code inside the same process without requiring a context switch. Proof of concept code showed this working with WebAssembly, which allowed an attacker to read data from arbitrary memory addresses and bypass memory sandboxing [49]. Additionally, the mitigations created for previous Spectre attacks do not affect SpectreRSB or ret2spec [45, 49].

#### 2.6.2.9 NetSpectre

NetSpectre adapts Spectre variant one into a generic, remote attack. Prior to this, all Spectre attacks required local code execution on the target system, so systems where attackers could not run code were thought safe. This attack has been shown to work both in local-area networks and between virtual machines in the cloud, and showed that even devices which do not run potentially attacker controlled code are now at risk for Spectre attacks [59].

#### 2.6.2.10 SplitSpectre

SplitSpectre is a variation of Spectre variant one. The original Spectre v1 requires a gadget to be present in the victim's attack surface. However, by splitting the original Spectre v1 gadget into two parts, SplitSpectre can be run within the attacker's own malicious code, instead of the target's kernel, simplifying the exploitation procedure and extending the length of the speculative execution window. A proof of concept of the attack has been created in Javascript [3].

**Figure 2.12: Microarchitectural Data Sampling Logo [56]**

### 2.6.3 Microarchitectural Data Sampling

Microarchitectural Data Sampling, MDS, is a class of CPU vulnerability that does not rely on assumptions about memory layout, or depend on the processor cache state. This separates them from Meltdown- and Spectre-like attacks; however, several MDS attacks are also counted as Meltdown-type attacks due to their use of fault or exception exploitation. Because of the lack of assumptions, MDS attacks are difficult to mitigate, though the structures involved are relatively small and are overwritten more frequently. Thus, benefiting from the exploit is difficult and requires an attack to collect a large amount of information in order to target a specific memory value. Using MDS, attackers can extract data from other programs on the same machine, across security boundaries, including SGX enclave boundaries. Like other speculative execution attacks, MDS based attacks do not leave evidence in system logs, making them effectively untraceable. Additionally, though using the vulnerability requires an attacker to have the ability to locally execute code, Javascript attacks have already been created [56]. Intel has stated that select 8th Gen and 9th Gen CPUs are already protected against the flaw, and that all future CPUs will include hardware mitigation. However, the researchers who discovered these flaws disagree and insist that the chips are still affected [67].

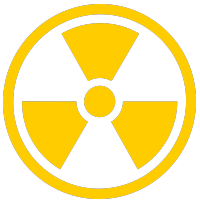


**Figure 2.13: Fallout Attack Logo [50]**

#### 2.6.3.1 Fallout

Fallout is both a Meltdown-type transient execution attack and a MDS attack. It leaks information from the store buffer by exploiting the Write Transient Forwarding optimization, which incorrectly passes values from memory writes to subsequent memory reads. The store buffer is used when a program writes to memory so that the program execution can continue while the virtual address is translated to a physical address. The processor must match the addresses of load instructions against the store buffer so that subsequent loads do not read stale values from memory. If a potentially stale value is found, it instead forwards the value from the store buffer. Write Transient Forwarding is used when a load partially matches a preceding store and the processor determines that it will fail because it could be in the store buffer, so it marks the load as faulty, and incorrectly forwards the value of the partially matched store from the store buffer. Using this, an attacker can reconstruct privileged information recently written by the kernel. As a Meltdown-type attack, Fallout exploits transient execution past an exception; however because the adversary does not read from the address of the protected value, and instead can load from an unrelated memory address in order to leak information, it is also an MDS attack. Because of this, not only do the current hardware countermeasures for Meltdown-type attacks not work against it, the newer Intel processors which were created to combat Meltdown and Spectre are instead more vulnerable than the older generations [50].

#### 2.6.3.2 Rogue In-Flight Data Load

Rogue In-Flight Data Load, RIDL, is a speculative execution attack that is able to leak arbitrary data across address spaces and privilege boundaries. It is not a variant of other speculative execution based attacks, i.e. Meltdown- or Spectre-based attacks. The cause of the vulnerability is micro-optimizations which cause the CPU to speculatively serve loads using extraneous CPU-internal in-flight data, e.g. in the line fill buffers. Because of this, RIDL can leak arbitrary in-flight data without assumptions about the cache or translation data structures, and can enable system-wide attacks from arbitrary unprivileged code, including browser based Javascript. RIDL is not affected by current short term mitigations in software and hardware, and cannot be easily mitigated by the current heavyweight defenses [72].

#### 2.6.3.3 Store-To-Leak Forwarding

Store-to-Leak Forwarding is a Meltdown-type attack that targets the store buffer and the TLB. The store buffer is a micro-architectural optimization which serializes the stream of stores, thus hiding the latency of storing values to memory. It allows the CPU to complete memory stores asynchronously while the execution stream is out of order. Due to its asynchronous nature, store-to-load forwarding can happen after an illegal memory store, thus allowing for Meltdown-like effects. This exploit can be used to mount side-channel attacks, break the atomicity of Intel's Transactional Synchronization Extensions, and monitor the control flow of the kernel. Additionally it can be used for several attacks on address space layout randomization, including but not limited to: an attack on kernel address space layout randomization, breaking out from unprivileged applications, and breaking address space layout randomization using Javascript [57].

**Figure 2.14: ZombieLoad Attack Logo [58]**

#### 2.6.3.4 ZombieLoad

Zombieload is a Meltdown-type attack which exploits a processor's fill-buffer logic. Load instructions that have to be reissued may transiently de-reference unauthorized destinations previously brought into the fill buffer by the current or shared logical CPU. This can leak the data of recently loaded stale values across logical cores [58]. Zombieload has been demonstrated to be effective in multiple practical attacks, including attacks across CPU privilege rings, OS processes, virtual machines, and SGX enclaves. Using this exploit, an attacker can see in real time the websites a victim is viewing, even if the victim is utilizing the TOR (The Onion Router) browser. An attacker could also acquire passwords, sensitive documents or encryption keys directly from a CPU [67].

### 2.6.4 Nemesis

Nemesis is a side channel attack that abuses the CPUs interrupt mechanism to leak microarchitectural instruction timings from enclaved execution environments. Although Nemesis uses similar microarchitectural behavior to Meltdown, i.e., exceptions and interrupts are delayed until instruction retirement, it is not a Meltdown attack because it does not use information from the faulting instructions but instead uses timing information. Using Nemesis, an attacker who is controlling system software can infer instruction-granular execution state from hardware-enforced enclaves

by measuring the latency of carefully timed interrupts. Unlike other speculative execution vulnerabilities, this attack is applicable to the whole computing spectrum. Proof of concept code showed interrupt latency revealing microarchitectural instruction timings from off-the-shelf Intel SGX enclaves [71].

### 2.6.5 TLBleed

TLBleed is an attack which utilizes machine learning in order to exploit the translation look-aside buffer, TLB, on processors which use simultaneous multithreading. It uses a timing attack on the TLB to gain fine-grained information about a victim, and is possible even when CPU cache uses side-channel protections. By using a machine learning strategy it can exploit high-resolution temporal features about a victim's memory activity in order to combat the unknown addressing functions inside the TLB. The proof of concept exploit can leak a 256-bit EdDSA secret key from a single capture after 17 seconds of computation time with a 98% success rate, even in the presence of state-of-the-art cache isolation. Similarly, using a single capture, the proof of concept can reconstruct 92% of RSA keys from an implementation that is hardened against FLUSH + RELOAD attacks [33].

### 2.6.6 Spoiler

Spoiler is a non-Spectre-based speculative execution attack that exploits the dependency resolution logic in order to gain information about the physical page mappings. This is done via a weakness in the address speculation of Intel's proprietary implementation of the memory subsystem which directly leaks timing behavior due to physical address conflicts. By taking advantage of this weakness, Spoiler is able to improve the Prime + Probe attack's eviction search by a factor of 4096, even from

sandboxed environments like Javascript. Additionally, Spoiler can be used to improve the Rowhammer attack by conducting DRAM row conflicts deterministically, and by demonstrating a double-sided Rowhammer attack with normal user's privilege due to the possibility of detecting contiguous memory pages [39].

This chapter was about some of the different microarchitectural optimizations used in modern processors, cache-timing attacks, and the recent high profile microarchitectural attacks that have caused performance decreasing mitigations to be implemented in the name of security. The next chapter covers the design of the experiment which investigated how much performance was lost when the mitigations to the microarchitectural attacks were enabled.

# Chapter 3

# DESIGN

In order to get a quantifiable impact of the software mitigations to the various microarchitectural attacks, the performance of many various programs should be looked at. These programs should cover a large spectrum of possible users, and not simply be a series of micro-benchmarks, as those are often optimized on by processor manufacturers. By using a wide variety of programs different implementations and usecases can be investigated. Testing on virtual machines allows for additional control conditions to be added and tested without the need to perform the tests on additional, expensive, hardware. Using a known test-bench software and associated tests allows for future comparisons to the same dataset.

## 3.1 Virtual Machines

Running the experiment on virtual machines allows for a comparison of both core count and vulnerability mitigation. The virtual machines will be different Ubuntu versions, see Table 3.1. Additionally, by comparing against all of the older versions of a release, the performance loss or gain from Linux development can be seen. It should be noted that all versions of Ubuntu 18 have been released since the discovery of Meltdown and Spectre, while version 16.04.04 on-wards have been influenced by their discovery.

**Table 3.1: Linux Distribution Patches and Release Dates**

| Linux Version | Release Date |
|---|---|
| 16.04.01 | 07 - 2016 |
| 16.04.02 | 02 - 2017 |
| 16.04.03 | 08 - 2017 |
| 16.04.04 | 02 - 2018 |
| 16.04.05 | 07 - 2018 |
| 16.04.06 | 03 - 2019 |
| 18.04.01 | 07 - 2018 |
| 18.04.02 | 02 - 2019 |

## 3.2 Phoronix Test Suite

The Phoronix Test Suite is a free, open-source benchmark software that supports Linux, Windows, Apple OS X, GNU Hurd, Solaris and BSD Operating Systems. It has access to more than 450 test profiles and over 100 test suites via OpenBenchmarking.org, and comes with built-in statistical result reporting. Tests and Test Suites inside Phoronix are built with eXtensible Markup Language.

## 3.3 Benchmarks

Traditionally, performance testing uses specific benchmark suites. This is so that when a new process is released, different agencies can use the same benchmark suites again and again in order to compare them on the same merits. However, this can lead to the problem of processors being optimized specifically for those benchmarks. If those benchmarks have enough depth and breadth this is not a problem, but too often similar sets of toy-program benchmarks are used. The following benchmarks

were chosen out of a compromise between time, environment, and function. All of the chosen benchmarks can run on both Windows and Linux, so that in the future the results can be compared with Windows results.

### 3.3.1 PostgreSQL

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (Transactions Per Second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction [35]. The various configurations that the Phoronix Test Suite has created can be seen in Table 3.2. Other options are possible by manually running pgbench; however, by using the pre-created options provided by Phoronix these results can be easily compared against both future tests and other tester's work.

**Table 3.2: pgbench Options**

| Scaling | Test | Mode |
|---|---|---|
| Buffer Test | Single Thread | Read Only |
| Mostly RAM | Normal Load | Read Write |
| On-Disk | Heavy Contention | |

**Table 3.3: pgbench Scaling Factors**

| Scaling Option | Scaling Factor |
|---|---|
| Buffer Test | 0.3% of System Memory |
| Mostly RAM | 20% of System Memory |
| On-Disk | 60% of System Memory |

**Table 3.4: pgbench Test Factors**

| Test Variant | Client Number | Thread Number |
|---|---|---|
| Single Thread | 1 | 1 |
| Normal Load | 4 * Number of CPU Cores | Number of CPU Cores |
| Heavy Contention | 16 * Number of CPU Cores | 2 * Number of CPU Cores |

#### 3.3.1.1 Scaling

The scaling factor in pgbench is the size of the database to test against. The default scaling factor of one corresponds to 100k rows in the main data table [35]. There are three different pre-created scaling options for the pgbench tests: Buffer Test, Mostly RAM, and On-Disk. In some places the Mostly RAM option is called the Mostly Cache option due to inconsistent naming. Each of the scaling options initializes pgbench with a different scaling factor, which can be seen in Table 3.3.

#### 3.3.1.2 Test

The number of clients is the number of concurrent database sessions. The thread number is the number of worker threads within pgbench. Clients are distributed as evenly as possible among available threads [35]. There are three different pre-created test variations: Single Thread, Normal Load, and Heavy Contention. Each uses different client and thread numbers in their testing, which can be seen in Table 3.4.

#### 3.3.1.3 Mode

The pgbench tests in Phoronix have two pre-defined modes, Read / Write, and Read Only. The Read / Write mode uses the default built-in transaction script. This script issues seven commands (made up of a selection from the UPDATE, SELECT, and

INSERT INTO commands) per transaction over randomly chosen numbers from the aid, tid, bid, and balance columns. If the Read Only mode is selected then only the SELECT is used.

### 3.3.2 DaCapo Benchmarks

The DaCapo benchmarks were created in order to see the complex interactions between the architecture, compiler, virtual machine, memory management, and application. They are written in Java because unlike C, C++, or Fortran, Java use stresses a machine's memory management and due to its virtual machine, will have more consistent benchmarks across different hardware [20].

#### 3.3.2.1 Eclipse

Eclipse is a free and open source integrated development environment and is the most widely used for Java programming. The Eclipse benchmark executes some of the non-graphical user interface Java Development Tool performance tests for the Eclipse IDE and measures how many milliseconds the tests take [8].

#### 3.3.2.2 H2

H2 is a relational database management system written in Java. It can be embedded in Java applications or run in client-server mode. The H2 benchmark executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application [8].

#### 3.3.2.3 Jython

Jython is an implementation of the Python programming language designed to run on the Java platform. The Jython benchmark interprets the pybench Python benchmarks [8]. The pybench benchmarks are a collection of tests created to provide a standardized way of measuring the performance of Python implementations. The tests inside pybench are micro-benchmarks which each test a different python functionality, e.g. string comparison, dictionaries with different keys, and simple arithmetic.

#### 3.3.2.4 DayTrader

DayTrader is an Open Source benchmark application emulating an online stock trading system. Both the TradeBeans and the TradeSoap benchmarks measure the time of the DayTrader benchmark. Both benchmarks use Apache Geronimo, an open source application server, and an in-memory H2 database.

The Tradebeans benchmark uses JavaBeans to connect to Apache Geronimo [8]. JavaBeans are classes that encapsulate many objects into a single object, allowing them to be serializable, have a zero-argument constructor, and allow access to properties using getter and setter methods.

The TradeSoap benchmark uses SOAP to connect to Apache Geronimo [8]. SOAP stands for Simple Object Access Protocol. It is an XML-based protocol for accessing web services.

### 3.3.3 SciMark

SciMark was developed by NIST and is widely used by the industry as a floating point benchmark. It is a benchmark for scientific and numerical computing. It consists of several subtests: Dense Lower-Upper Matrix Factorization, Fast Fourier Transform, Jacobi Successive Over-Relaxation, Monte Carlo, and Sparse Matrix Multiply. There are two versions of this test, one with a "large" dataset, 32 MB, which stresses the memory subsystem and a "small" dataset which stresses the JVMs, 512 KB. The units in SciMark 2.0 are MFLOPs (Millions of floating point operations per second) because some of the benchmarks exercise transcendental functions (e.g. sin, cos) or integer operations. Therefore, the MFLOP count is only approximate; however, the same MFLOP count is used consistently, to ensure that comparisons are valid. SciMark 2.0 focuses only on single-processor performance [53].

This thesis uses both the Java and ANSI C versions of SciMark 2.0. The ANSI C version uses the "large" dataset, and the Java version the "small" dataset [13, 10].

#### 3.3.3.1 Dense LowerUpper Matrix Factorization

The Dense LowerUpper (LU) matrix factorization benchmark computes the LU factorization of a dense 100x100 matrix using partial pivoting. It exercises linear algebra kernels and dense matrix operations. The algorithm is the right-looking version of LU with rank-1 updates. The data size for the "large" version of the benchmark uses a 1,000 x 1,000 matrix [53].

#### 3.3.3.2 Fast Fourier Transform

The Fast Fourier Transform benchmark performs a one-dimensional forward transform of 4,000 complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions. The first section performs the bit-reversal portion (no flops) and the second performs the actual Nlog(N) computational steps. The data size for the "large" version of the benchmark is $2^{20}(= 1048576)$ complex numbers [53].

#### 3.3.3.3 Jacobi Successive Over-Relaxation

This benchmark performs a Jacobi Successive Over-Relaxation on a 100x100 grid. It exercises a typical access pattern in finite difference applications – solving Laplace's equation in 2D with Drichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns, where each A(i,j) is assigned an average weighting of its four nearest neighbors. Some hand-optimizing is done to rows so that they are aliased in order to streamline the array accesses in the update expression. The data size for the "large" version of the benchmark uses a 1,000 x 1,000 grid [53].

#### 3.3.3.4 Monte Carlo

The Monte Carlo benchmark approximates the value of Pi by computing the integral of the quarter circle $y = sqrt(1-x^2)$ on [0,1]. It chooses random points and computes the ratio of those within the circle. The algorithm exercises random-number generators, synchronized function calls, and function inlining. Because this benchmark uses only scalars, the "large" and "small" versions are identical [53].

#### 3.3.3.5 Sparse Matrix Multiply

The Sparse Matrix Multiply benchmark uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirect addressing and non-regular memory references. The "small" version uses a 1,000 x 1,000 sparse matrix with 5,000 nonzeros. Each row has approximately 5 nonzeros, evenly spaced between the first column and the diagonal. The "large" version uses a 100,000 x 100,000 matrix with 1,000,000 nonzeros [53].

#### 3.3.3.6 Composite

The composite benchmark reruns all of the above benchmarks again in order to get an average MFLOPs number. This number is often used when SciMark is used as an independent benchmark suite.

### 3.3.4 Encoding

Encoding is the process of putting a sequence of characters (letters, numbers, punctuation, and certain symbols) into a specialized format for efficient transmission or storage. This is done through code in order to change original data into a form that can be used by an external process. Encoding is also used to reduce the size of audio and video files. Each audio and video file format has a corresponding coder-decoder (codec) program that is used to code it into the appropriate format and then decodes for playback.


**Figure 3.1: Screenshot from x264 Test Sample Video**

#### 3.3.4.1 LAME MP3 Encoder

LAME is a free and open source MP3 encoder licensed under the LGPL. The name LAME is a recursive acronym for LAME Ain't an Mp3 Encoder, presumably due to the potential software patent issues. The goal of the LAME project is to use the open source model to improve the psycho acoustics, noise shaping, and speed of MP3. Although LAME is distributed only as source code, it is used inside popular software such as FFmpeg, Audacity, WinAmp and Blaze Media Pro. [54] This test measures the time required to encode a seven minute WAV file to MP3 format [18].

#### 3.3.4.2 x264

x264 is a free and open-source software library and a command-line utility developed by VideoLAN for encoding video streams into the H.264/MPEG-4 AVC format. It has SIMD assembly code acceleration on the x86, PowerPC, and ARMv7 platforms. It has support for different applications, such as television broadcast, Blu-ray low-latency video applications, and web video and is the core of many web video services, such as Youtube, Facebook, Vimeo, and Hulu. It is widely used by television broadcasters and

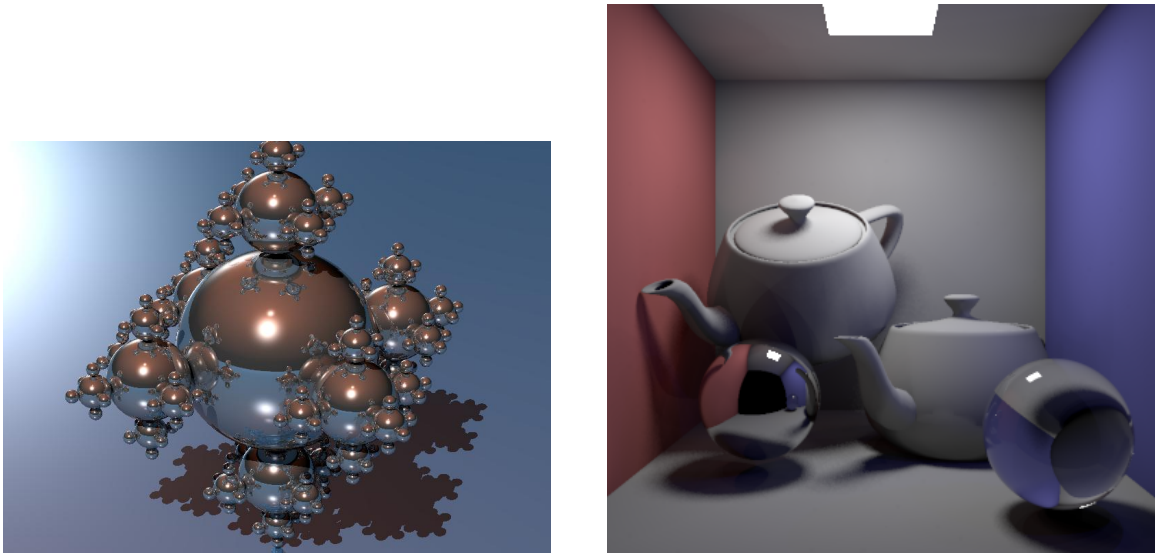
ISPs. Multiple common video encoding programs are frontends for x264, including HandBrake, Avidemux, FFmpeg, and MediaCoder. [51] The test of the x264 encoder tests only the CPU by disabling OpenCL support. It measures how long it takes to encode a video file [17]. The test's video file is used under the Creative Commons license, and is twenty-one seconds of the scene in Figure 3.1.

### 3.3.5 Ray Tracing

Ray tracing is a technique for rendering three-dimensional graphics with very complex light interactions, e.g. reflection, refraction, scattering, and dispersion phenomena. It generates an image by tracing the path of light as pixels in an image plane and then simulating the effects of that path when it encounters other virtual objects. Although it has a high computational cost, it is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods. [55] Therefore, ray tracing is best suited for applications where taking a relatively long time to render a frame can be tolerated, such as in still images and film and television visual effects. It is poorly suited for real-time applications, such as video games, where speed is critical.

#### 3.3.5.1 C-Ray

C-Ray is a simple raytracer designed to test the floating-point CPU performance. It was created because a more advanced raytracer would have more overhead for disk I/O, shader parsing, and more strain on the memory bandwidth, but would still be FPU-limited because of the floating-point math. Therefore, being able to test the bottleneck of the performance without the additional overhead was desirable. The test is multi-threaded, using 16 threads per core, and shoots 8 rays per pixel [7]. It



(a) C-Ray sphfract

(b) Sunflow Rendering System

**Figure 3.2: Ray Tracing Generated Images**

measures the generation time of the 1600 x 1200 image – Figure 3.2a.

#### 3.3.5.2 Sunflow Rendering System

The Sunflow Rendering System is a rendering system for photo-realistic image synthesis. It is written in Java and built around a flexible ray tracing core and an extensible object-oriented design [15]. The test runs a benchmark from the Sunflow Rendering System, in which it creates the image in Figure 3.2b several times and measures the speed of creation.

### 3.3.6 Compression

Data compression reduces the amount of space needed to store files. When the size of a file is halved, twice as many files can be stored for the same cost and the download speed is twice as fast. When dealing with a large amount of files, halving the amount of

space taken can represent a massive reduction in the space and computing required. This leads to a large savings in power consumption and cooling which is a huge reduction in the impact on the environment.

In order to get these benefits, a compression program needs processing time to compress and decompress the data. The faster this processing can be done, the more efficient the savings from compression are.

#### 3.3.6.1 7-Zip Compression

7-Zip is free, open source software. It creates compressed .7z files using LZMA and LZMA2 compression [2]. The 7-Zip Compression test uses p7zip, the Linux port of 7-zip. It measures how many MIPS (millions of instructions per second) for a compression of the Windows x64 build using its integrated benchmark feature [6].

#### 3.3.6.2 Gzip Compression

Gzip is a free, open source software included with most Linux distributions. It creates .tar.gz files using the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding [32]. The gzip test uses the tar utility included in most Linux distributions. This test measures the time needed to archive/compress two copies of the Linux 4.13 kernel source tree when using the *tar -zcf* Linux command [9]. The *-z* is the option for *tar* to use gzip.

### 3.3.7 Miscellaneous

These are tests that, while useful and informative, did not fit well into any of the previous categories.

#### 3.3.7.1 Bork

Bork is a small, cross-platform file encryption utility originally written in 2004. It is written in Java and uses a stream cipher with RC4, and is able to obfuscate filenames with SHA-1 hashing. RC4 and SHA-1 are now considered insecure [73, 64]. It was designed to be included along with the files it encrypts for long-term (e.g. on CD) storage, with minimal dependencies in the attempt to make it future proof. This test measures the amount of time it takes to encrypt a sample file [5].

#### 3.3.7.2 MAFFT

MAFFT is a free program used to create multiple sequence alignments of amino acid or nucleotide sequences. MAFFT offers multiple alignment strategies, with tradeoffs between speed and accuracy [41]. This test performs an alignment of 100 pyruvate decarboxylase sequences, with pairwise alignments computed with the Smith-Waterman algorithm and 20000 iterative refinement cycles performed [16].

#### 3.3.7.3 R Benchmark

The R Benchmark in the Phoronix Test Suite downloads several R benchmarks; however, only R-benchmark-25, ATT benchmark, is used. The Phoronix Test Suite version is customized for rbench driver [11]. R-benchmark-25 is a series of R benchmarks with three categories: matrix calculation, matrix functions, and programmation. The matrix calculation benchmarks include: the creation, transformation and deformation of a 2500x2500 matrix, sorting of 7,000,000 random values, the cross-product a 2800x2800 matrix against itself, and the linear regression of a 3000x3000 matrix. The matrix function benchmarks include: a FFT over 2,400,000 random values, a calcula-

**Figure 3.3: Screenshot of Unigine - Sanctuary**

tion of the eigenvalues of a 640x640 random matrix, the determinant of a 2500x2500 random matrix, the Cholesky decomposition of a 3000x3000 matrix, and the inverting of a 1600x1600 matrix. The programmation benchmarks include: a calculation of 3,500,000 Fibonacci numbers, the creation of a 3000x3000 Hilbert matrix, the recursive calculation of the grand common divisors of 400,000 pairs, the creation of a 500x500 Toeplitz matrix, and a calculation of Escoufier's method on a 45x45 matrix. The end result is the mean of each categories trimmed geometric mean [34].

#### 3.3.7.4 SQLite

This is a simple benchmark of SQLite. SQLite is a software library that provides a relational database management system. The lite in SQLite means light weight in terms of setup, database administration, and required resources. SQLite has the following noticeable features: self-contained, serverless, zero-configuration, and transactional [62]. This test profile measures the time to perform a pre-defined number of insertions on an indexed database [14].

#### 3.3.7.5 Unigine - Sanctuary

Unigine is a proprietary cross-platform game engine, developed by Russian software company Unigine Corp. A trial version of the engine, called the "Evaluation Kit," is provided to companies working on commercial projects [69]. This test calculates the average frame-rate within the Sanctuary demo for the Unigine 1 engine. The Sanctuary demo has five dynamic lights, high dynamic range rendering, parallax occlusion mapping, ambient occlusion mapping, volumetric light and fog, and particle systems [68]. It is an interactive experience with fly-through mode of the scene in Figure 3.3. This test includes both a windowed and a fullscreen mode with a resolution of 800 x 600. [12].

This chapter was about the design of the experiment, how it was setup, and which tests were used as benchmarks and why. The next chapter covers how the experiment was setup, details about the hardware and virtual machines used, and information about the software mitigations.

# Chapter 4

# IMPLEMENTATION

To gather the data for the analysis all of the benchmarks from the previous chapter were run at least once where they gathered a minimum of three data points per run. Most benchmarks were run an additional time after the initial round of testing, in order to increase the number of data points per test; however, because of how long the pgbench benchmarks took (about 17 hours for Ubuntu 16 and about 28 hours for Ubuntu 18) a full second run of that benchmark could not be completed. Tests were completed for the different versions of Ubuntu 16 and 18, allowing for a comparison of 32-bit to 64-bit. Additionally, comparing across the different version numbers demonstrates what performance loss is normal from a version update vs. the mitigations to speculative execution attacks. Using the latest versions of Ubuntu 16, 16.04.06, and Ubuntu 18, 18.04.02, the affect of having or not having the default mitigations was across several core counts. This allowed for a view into the performance affect of the mitigations on each core level and the affect of having more or less cores for each benchmark.

The tests were completed on virtual machines with an Intel Haswell processor. The mitigations were only tested in their default and "off" state, and mitigations that required a kernel recompile were ignored.

**Table 4.1: Virtual Machine Information**

| Operating System | Memory | Video Memory | Processors | Hard Drive Size |
|---|---|---|---|---|
| 16.04.01 | 8192 MB | 128 MB | 2 Cores | 25 GB |
| 16.04.02 | 8192 MB | 128 MB | 2 Cores | 25 GB |
| 16.04.03 | 8192 MB | 128 MB | 2 Cores | 25 GB |
| 16.04.04 | 8192 MB | 128 MB | 2 Cores | 25 GB |
| 16.04.05 | 8192 MB | 128 MB | 2 Cores | 25 GB |
| 16.04.06 | 8192 MB | 128 MB | Variable | 50 GB |
| 18.04.01 | 8192 MB | 128 MB | 2 Cores | 35 GB |
| 18.04.02 | 8192 MB | 128 MB | Variable | 150 GB |

## 4.1 Virtual Machines

Oracle VM Virtual Box, version 6.0.4 r128413 (Qt5.6.2), was used to create and manage the virtual machines. Information about these machines can be seen in Table 4.1. Ubuntu 18 required a larger amount of hard drive space than Ubuntu 16 due to it being a default 64-bit operating system, unlike Ubuntu 16 which is a default 32-bit. Additionally, because of its 64-bit status, gcc-multilib had to be installed for some tests, meaning that some tests could natively use 64-bit while others could not. Ubuntu 18.04.02 required a very large hard drive; tests with smaller hard drive sizes (25GB, 35GB, 50GB, 75GB and 100GB) failed pgbench runs due to a size error. Ubuntu 16.04.06 had a similar issue, but the error resolved with a 50GB hard drive rather than 18.04.02's 150GB.

## 4.2 Host Machine

The machine used to host these tests is a Windows 10 Pro Desktop computer. It has 16GB of RAM and all of the virtual machines were run off of a 2TB hard drive.

### 4.2.1 CPU

The CPU used in these tests is an Intel Haswell i5-4690K. It has a clock speed of 3.50GHz and 4 cores. The Haswell micro-architecture was released in 2013 and is the successor to the Sandy and Ivy Bridge micro-architectures. It has improvements in the out-of-order scheduling, execution units, and memory hierarchy, and is the first instance of Intel supporting Fused Multiply-Add operations. Additionally, several new dispatch and memory ports were added, and several of the instruction ports were enlarged to 256-bits, thus allowing for both increased throughput and pipeline efficiency. The largest improvements that the Haswell can boast are to the memory hierarchy. Its L2 TLB was substantially improved by adding support for 2MB pages and giving it twice the number of entries. Similarly, the L1 data cache was increased in bandwidth by a third, allowing it to sustain two 256-bit loads and a 256-bit store every cycle [40].

### 4.2.2 OS Patch

On April 11th a power failure caused the host machine to update and apply the KB4493464 OS patch. At this time the first round of Ubuntu 16 patch tests were finished and the core test changes had started, with only the 2 cores and 2 cores vulnerable tests completed. This patch contained mitigations for Meltdown and Spectre Variant 2 for VIA-based computers. and while this machine is does not have a VIA

**Table 4.2: Vulnerabilities and Mitigations**

| Vulnerability | 16.04.04 | 16.04.05 | 16.04.06 | 18.04.01 | 18.04.02 |
|---|---|---|---|---|---|
| Meltdown | Default Off | Default Off | Default Off | Default On | Default On |
| Foreshadow | Not Available | Always On | Always On | Always On | Always On |
| Spectre v1 | Always On | Always On | Always On | Always On | Always On |
| Spectre v2 | Default On | Default On | Default On | Default On | Default On |
| Speculative Store Bypass | Not Available | Default Off | Default Off | Default Off | Default Off |

CPU, it appears to have affected the Mostly RAM pgbench tests drastically. It is possible, though unlikely, that there are VIA chips in the motherboard, an ASUS Z97-A, that caused the machine to be affected; however the manufacturer states that it uses the Intel Z97 Express Chipset [4].

## 4.3 Mitigations

The mitigations in this section are those that were visible when the testing was taking place. These are displayed in the `/sys/devices/system/cpu/vulnerabilities/` directory, where name of the file is the vulnerability and the contents either `Vulnerable` or the name of the mitigation. Table 4.2 shows their default state and history for the two different operating systems. Other vulnerabilities and attacks either were not addressed yet by Linux or were fixed entirely by microcode updates and therefore were not visible as a specific mitigation. Testing was between the specific operating system's default state and a state where all mitigations that could be disabled without a kernel recompile, were. It is important to note that past versions of Ubuntu 16 and 18 also have some of the mitigations enabled.

### 4.3.1 GRUB

GRUB is a boot loader for Linux that allows a user to select a specific kernel configuration. The options for GRUB are found in `/etc/default/grub`.
`GRUB_CMDLINE_LINUX_DEFAULT` is used to change which mitigations are enabled. When this file is changed, the system can be commanded to update grub and reboot, and the resulting machine will have the default mitigations disabled.

### 4.3.2 Meltdown

The Meltdown vulnerability (CVE-2017-5754) can be completely mitigated in software [47]. This mitigation is kernel page-table isolation (KPTI), previously called KAISER.

KPTI works by separating kernel and user space into two different sets of page tables. Kernel page tables maintain their implementation from before KPTI was introduced, in which the page tables contain both kernel-space and user-space addresses. However, this implementation is only used when the system is running the kernel. The user has a different set of page tables, where there is a copy of user-space and a minimal set of kernel-space mappings. The kernel mappings in the user page tables are required if the process needs to enter or exit system calls, interrupts and exceptions. When leaving user-space for kernel-space, or vice versa, the translation lookaside buffer (TLB) is flushed [25]. Newer processors with support for the process-context identifiers (PCID) can avoid this; however, there is still a significant performance cost, especially in syscall-heavy and interrupt-heavy workloads.

KPTI can partially be disabled with the *nopti* kernel boot option. Ubuntu 16.04.06 does not have KPTI enabled by default, while Ubuntu 18.04.02 does.

### 4.3.3 Foreshadow

Intel has classified all of the Foreshadow vulnerabilities (CVE-2018-3615, CVE-2018-3620, CVE-2018-3646) as Level One Terminal Fault, L1TF, and they can be partially mitigated in software. The most basic mitigation is Page Table Entry (PTE) Inversion. By inverting all of the bits in a PTE when it is not marked as present the PTE will point to a nonexistent region of memory [26]. This is enabled by default and cannot be turned off without a kernel recompile.

The two additional, non-default, mitigations are only if Kernel Virtual Machine (KVM) is enabled. The first turns off symmetric multithreading (SMT), or hyperthreading on Intel machines. Disabling SMT can have a significant performance impact, and therefore must be weighted against the impact of other mitigation solutions like confining guests to dedicated cores. The other is to have the hypervisor flush the L1 Data Cache (L1D) before entering the guest. Flushing the L1D evicts both guest data and any data that should not be accessed by malicious guests. However, this also has a performance impact as the processor has to bring the flushed guest data back into the L1D [42]. This thesis uses Ubuntu as the Guest machine, and not the host, so the virtual machines do not have KVM enabled. Thus, these two mitigations can be ignored.

### 4.3.4 Spectre Variant 1

Spectre Variant 1 (CVE-2017-5753), can be mitigated in software only by patching code sequences found to be vulnerable. To patch a vulnerable section, a barrier instruction, LFENCE, is inserted in order to stop speculation. Alternatively, all instructions can be serialized in order to stop younger instructions from executing, even speculatively, before older instructions have retired, but LFENCE is a better

performance solution. An LFENCE instruction inserted after a bounds check will prevent younger operations from executing before the bound check retires. However, it must be used carefully; if used too often, performance is compromised. Developers are using static analysis to find these vulnerable sections, but if even one section in a codebase is skipped the entire codebase is still vulnerable [38]. This mitigation is enabled by default on operating systems that have it, because it is compiled into the kernel. If a user decompiles and recompiles the kernel it can be turned off, but for this thesis it was not.

### 4.3.5 Spectre Variant 2

Google developed the Full Generic Retpoline mitigation for Spectre Variant 2 (CVE 2017-5715). A retpoline is a "return trampoline", where the software replaces indirect near jump and call instructions with a code sequence that includes pushing the target of the branch in question onto the stack and then executing a return instruction to jump to that location, as return instructions can be protected using this method [38].

### 4.3.6 Speculative Store Bypass

The mitigation for Speculative Store Bypass (CVE 2018-3639) is not on by default on either 16.04.06 or 18.04.02. This is because Intel recommends enabling this mitigation only for managed runtimes or other situations that use language-based security to guard against attacks within an address space. If this mitigation is enabled the system sets the Speculative Store Bypass Disable bit in order to prevent loads from executing before all older store addresses are known. Software that does not use language-based security should instead carefully insert LFENCE instructions, insert additional register dependencies between vulnerable loads and stores, or isolate secrets

into a separate address space from code that is relying on language-based security [38]. For the purpose of this thesis this mitigation was ignored because it was not on by default.

This chapter was about how the experiment was setup, details about the hardware and virtual machines used, and information about the software mitigations. The next chapter presents both an analysis of and the results of the experiment.

# Chapter 5

# RESULTS

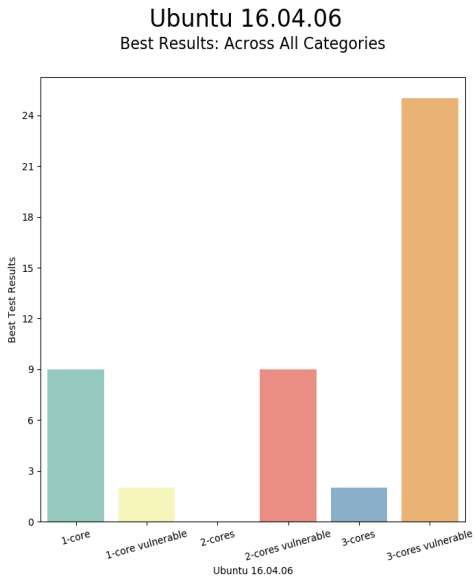
Most benchmarks behaved as expected, with test runs with more cores outperforming those with less, and tests with mitigations disabled outperforming those with them enabled. While some tests demonstrate outliers these are all within testing runs that have otherwise consistent data. Since some of these outliers appear across multiple testing runs on different dates it was considered safe to leave them in as merely unusual but acceptable data points. The exception to this is the Host Patch tests, where the data before and after the patch to the Host machine was so drastically different that it was separated out as its own graph set, and not included in the averages.

**Table 5.1: Average Percent Difference Per Core Count**

| Core Number | 16.04.02 Core | 18.04.02 Core |
|---|---|---|
| 1 core | 3.40% | 2.54% |
| 2 cores | 3.07% | 4.93% |
| 3 cores | 3.47% | 3.68% |

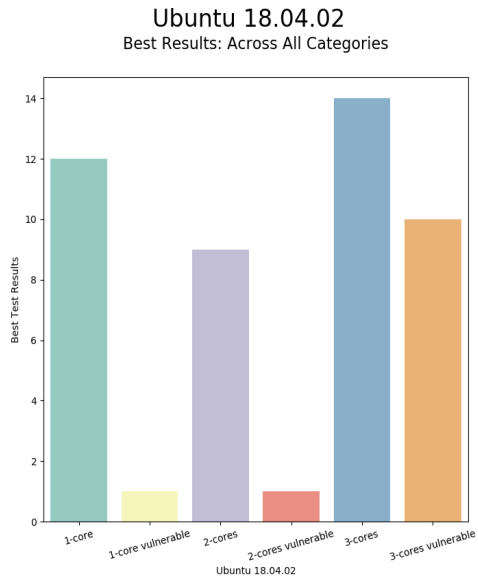
## 5.1 Average Performance

Looking at the average performance of the tests prevents outliers from skewing the data. Table 5.1 compares the average percent difference for vulnerable versus mitigated tests within the same core count, showing that the average variance between tests was low. Additionally, tests were compared across both patch and core number to see which had the best results, with best being determined by each benchmark because some for benchmarks higher numbers are better and others, lower.




(a) Ubuntu 16.04.06

Best Results: All Categories



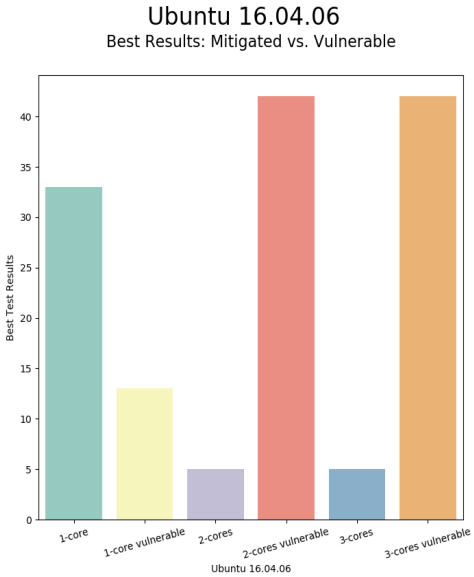

(b) Ubuntu 18.04.02

Best Results: All Categories

**Figure 5.1: Best Results: All Categories, Core Number and Vulnerabilities**

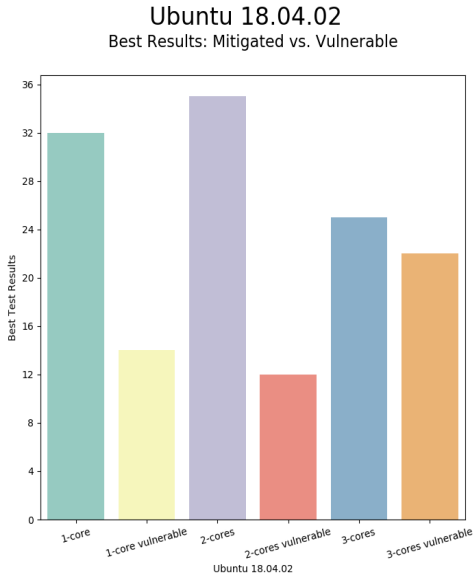
Figure 5.1 compares the results of each benchmark for all of the possible core counts. It shows that for Ubuntu 16, 3 cores vulnerable is almost always the best, while for Ubuntu 18 the results are more spread out. The Ubuntu 18 results may be affected by the patch to the host machine, and on the benchmarks where this looks possible it is called out.

Figure 5.2 compares the results by looking only at each core level, in order to examine whether mitigations had an impact. Interestingly, for Ubuntu 16, when there is more than one core the vulnerable version is almost always better. Likely because many of the mitigations involve both multiple physical and logical cores; enabling those mitigations makes those tests slower. Thus, the mitigations don't have much effect when tested on only one core. As before in figure 5.1, the Ubuntu 18 results are spread out across all categories. Notably no vulnerable category outperforms its default counterpart, potentially due to the patch to the host machine.




(a) Ubuntu 16.04.06

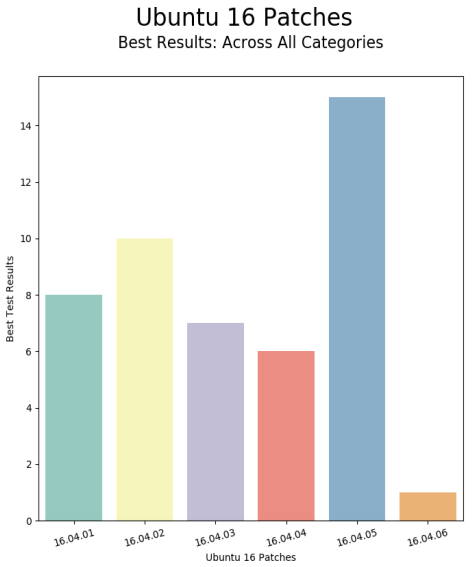
Best Results: Mitigated vs. Vulnerable




(b) Ubuntu 18.04.02

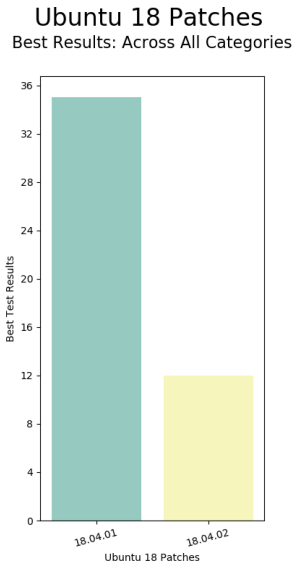
Best Results: Mitigated vs. Vulnerable

**Figure 5.2: Best Results: Mitigated vs. Vulnerable**




(a) Ubuntu 16

Best Results: All Categories




(b) Ubuntu 18

Best Results: All Categories

**Figure 5.3: Best Results: All Categories, Patch Level**

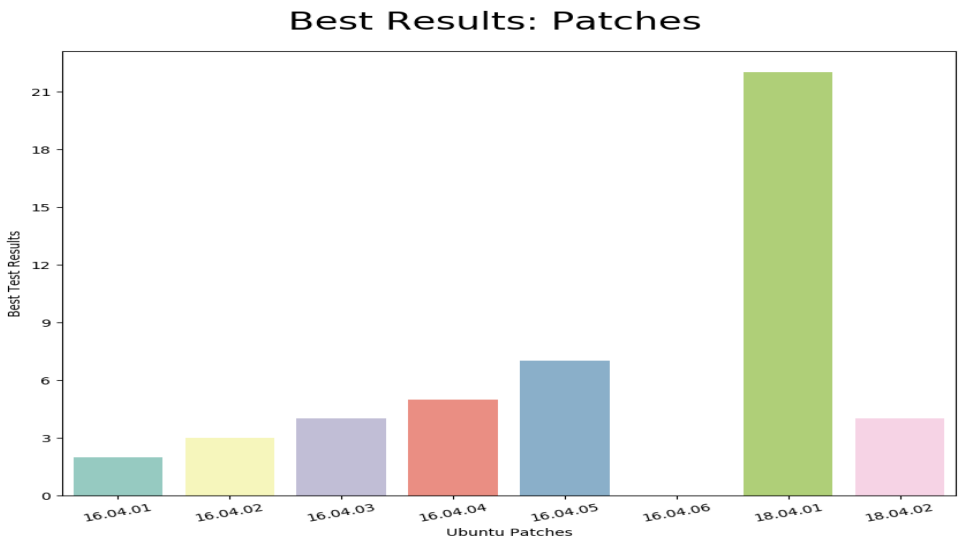



**Figure 5.4: Best Results: Patches**

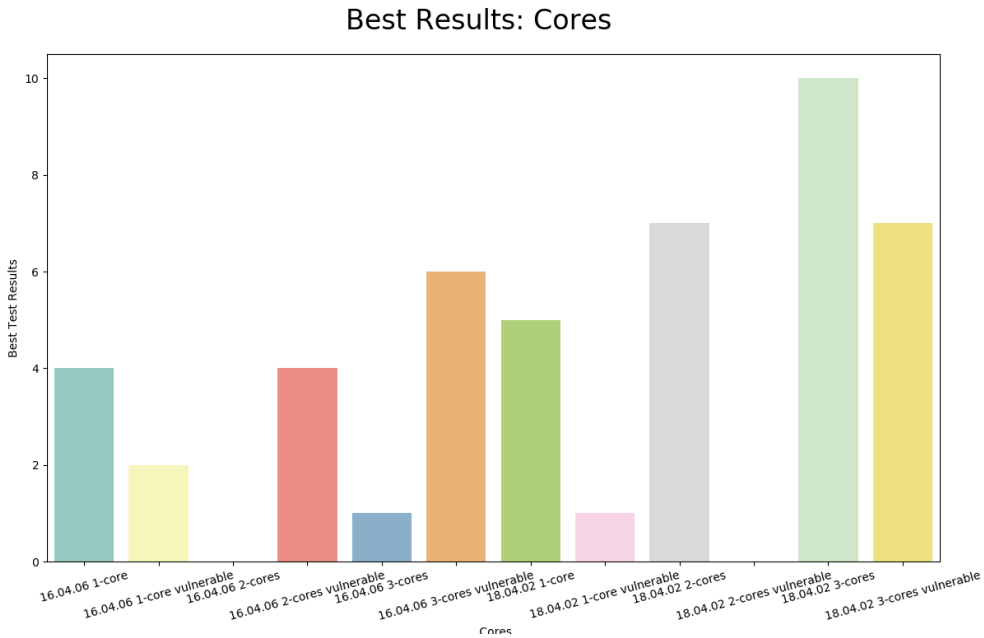



**Figure 5.5: Best Results: Cores**

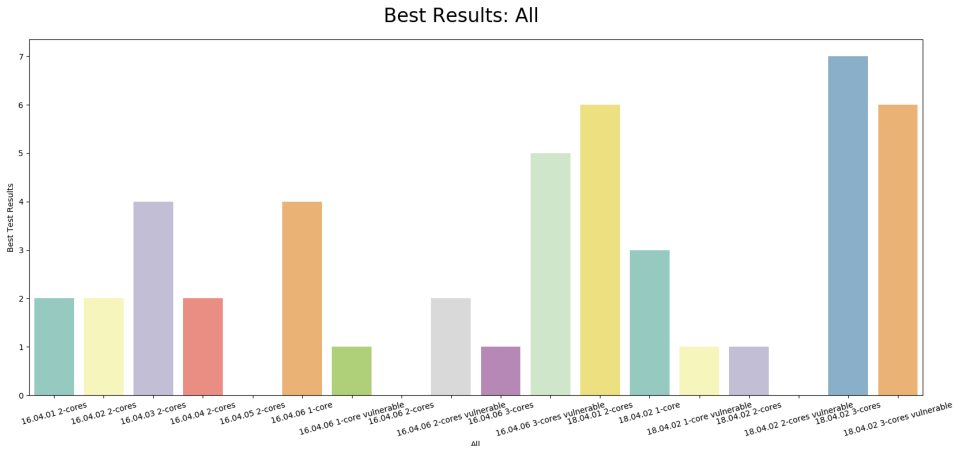



**Figure 5.6: Best Results: All**

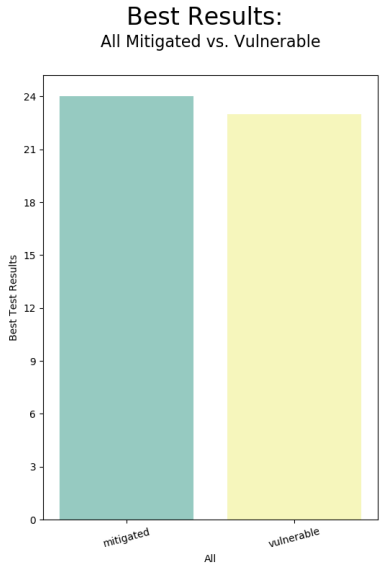



**Figure 5.7: Best Results: All**

Figure 5.3 compares the results only across different patch levels within their version. Doing this reveals that the best Ubuntu 16 version appears to be 16.04.05, which has the same default mitigations as 16.04.06. Likewise, Ubuntu 18.04.01 has the same mitigations enabled by default as its competitor. This could mean that the large gulf between the patch levels is due to the additional micro-op patches for unstated vulnerabilities.

All patch results can be seen in figure 5.4, where it can be seen that Ubuntu 18.04.01 is the best version when using two cores and default mitigations. This is unexpected because by default Ubuntu 18 has one more mitigation, kpti, turned on than Ubuntu 16. Therefore, this must be a combination of Ubuntu 18 being a 64-bit native operating system and benchmarks having adapted to that.

Figure 5.5 compares the results from all tests that changed the number of cores and disabled mitigations. This shows that Ubuntu 18.04.02 with 3 cores is the best, followed by two more 18.04.02 results with different core levels. Ubuntu 16.04.06 with 3 cores vulnerable does not appear until after these two results, thus again showing how powerful the 64 bit operating system is.

In figure 5.6 the result of all tests compared against each other can be seen. Again, Ubuntu 18 is shown to be superior, and interestingly 18.04.01 can almost compete with 18.04.02 even when it is at one less core. Like figure 5.4 ,this emphasizes both the power of a native 64-bit operating system, even when the Meltdown mitigation, kpti, is different between Ubuntu 16 and 18. It also shows how brutal the 18.04.02 micro-op changes are, since the earlier patch can compete with a later version utilizing an additional core.

Lastly, figure 5.7 shows only mitigated versus vulnerable results compared. The vulnerable results are the explicitly vulnerable Ubuntu 16.04.06 and 18.04.02 tests

and the versions of Ubuntu from before Meltdown- and Spectre-type attacks emerged, specifically 16.04.01, 16.04.02, and 16.04.03. The mitigated results are the default Ubuntu 16.04.06 and 18.04.02 tests as well as the other Ubuntu 16 and 18 patches. Here it can be seen that the number of tests that do better without mitigations is almost equal to the number that do better with mitigations. This shows that while many tests are affected by the software mitigations, it is not the overwhelming majority. Thus, if a user wanted to have better performance they would need to research their usecase specifically, and could not necessarily get a performance increase by removing their mitigations.

## 5.2 PostgreSQL

The pgbench Benchmarks have the most diversity when looking at the effect of both patch and core differences. They are almost universally better with a higher core number and with mitigations disabled. Additionally, these benchmarks are extremely affected by the patch to the host machine, as seen by figures 5.8, 5.9 and 5.10. Because of this massive disparity in performance, data from before and after the patch has been separated.

**Table 5.2: Buffer Tests Benchmark Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Heavy Contention - RO | 16.04.01 | 3 cores vulnerable | 18.04.01 | **3 cores vulnerable** |
| Heavy Contention - RW | 16.04.04 | 3 cores vulnerable | 18.04.01 | **3 cores** |
| Normal Load - RO | 16.04.01 | 3 cores vulnerable | 18.04.01 | **3 cores vulnerable** |
| Normal Load - RW | 16.04.05 | 3 cores vulnerable | 18.04.02 | **3 cores vulnerable** |
| Single Thread - RO | 16.04.03 | 1 core | 18.04.01 | **1 core vulnerable** |
| Single Thread - RW | **16.04.04** | 2 cores vulnerable | 18.04.01 | 1 core |

**Table 5.3: Buffer Tests Benchmark Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Heavy Contention - RO | 8.02 | 97.96 | 19.67 | 104.96 |
| Heavy Contention - RW | 43.98 | 59.12 | 5.32 | 74.58 |
| Normal Load - RO | 12.55 | 98.41 | 18.55 | 106.23 |
| Normal Load - RW | 43.71 | 65.77 | 2.44 | 73.66 |
| Single Thread - RO | 27.89 | 57.21 | 7.09 | 61.08 |
| Single Thread - RW | 17.25 | 10.52 | 11.88 | 3.34 |

### 5.2.1 Buffer Test

Table 5.2 shows that for the tests that use additional cores, it is almost always better to have the mitigations disabled. Interestingly, the buffer tests are the only tests where Ubuntu 18 has better performance than Ubuntu 16, likely due to the smaller scale factor making up for missing optimizations. Table 5.3 shows large discrepancies between the best and the worst for each category. Looking at figures A.7, A.8, A.9, A.10, A.11, and A.12 this appears to be from certain tests having high degrees of variance due to unusual outliers; however because they are single datapoint outliers in otherwise normal tests, the outlier could be the initial cache miss.

**Table 5.4: RAM Tests Benchmark Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Heavy Contention - RO | **16.04.03** | 2 cores vulnerable | 18.04.02 | 2 cores |
| Heavy Contention - RW | **16.04.02** | 3 cores vulnerable | 18.04.02 | 3 cores |
| Normal Load - RO | **16.04.03** | 3 cores | 18.04.02 | 3 cores |
| Normal Load - RW | **16.04.01** | 3 cores vulnerable | 18.04.02 | 3 cores vulnerable |
| Single Thread - RO | **16.04.03** | 1 core | 18.04.02 | 2 cores |
| Single Thread - RW | **16.04.03** | 3 cores vulnerable | 18.04.02 | 2 cores |

**Table 5.5: RAM Tests Benchmark Percent Difference of Average**

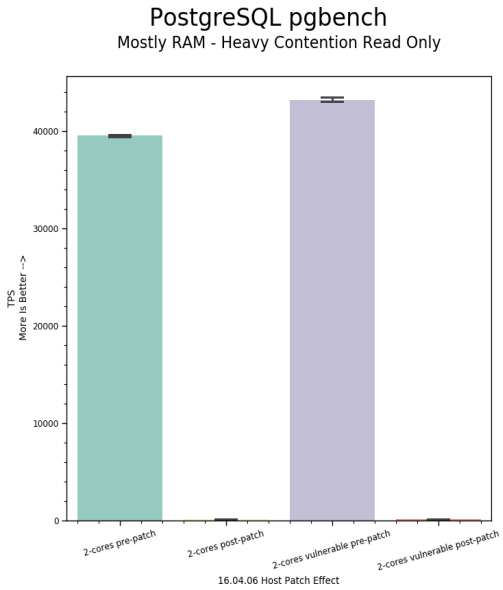
| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Heavy Contention - RO | 7.09 | 9.94 | 199.77 | 14.31 |
| Heavy Contention - RW | 3.24 | 18.26 | 199.71 | 21.14 |
| Normal Load - RO | 8.81 | 41.19 | 199.86 | 45.58 |
| Normal Load - RW | 5.09 | 24.70 | 199.88 | 27.68 |
| Single Thread - RO | 38.03 | 11.51 | 199.97 | 10.54 |
| Single Thread - RW | 17.99 | 6.86 | 199.98 | 20.16 |

### 5.2.2 Mostly RAM

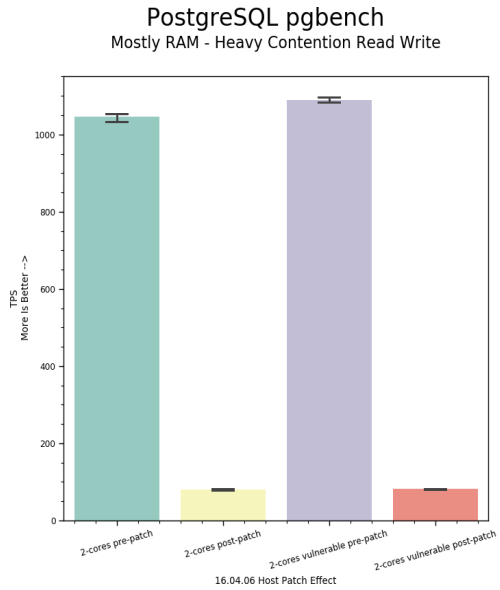
Table 5.4 shows that earlier versions of Ubuntu 16 perform better. Additionally, 16.04.06 performs better with more cores, showing that the mitigations are more powerful than the core number for this benchmark. Within both Ubuntu 16.04.06 and 18.04.02 higher core number tests almost always win, especially when mitigations are disabled. Although table 5.5 would indicate that the percent difference between core levels is massive, the figures A.13, A.14, A.15, A.16, A.17, and A.18 show that these tests share the relationship of more cores performing markedly better. These figures also illustrate the massive discrepancy between 18.04.01 and 18.04.02.

### 5.2.3 Mostly RAM - Host Patch

The Mostly RAM test is the most dramatically affected by the application of the patch to the host system. Figures 5.8, 5.9, and 5.10 show the order of magnitude difference between the pre- and post-patch versions. Bizarrely, according to the documentation of the patch, this should have effected only VIA-based systems, which the host machine is not. Further research into if Meltdown and Spectre mitigation patches for other chip types are inadvertently affecting unrelated machines is needed.

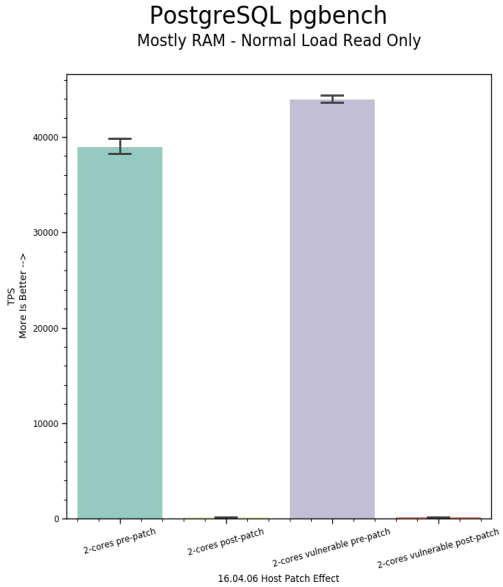



(a) 16 Cores - Read Only

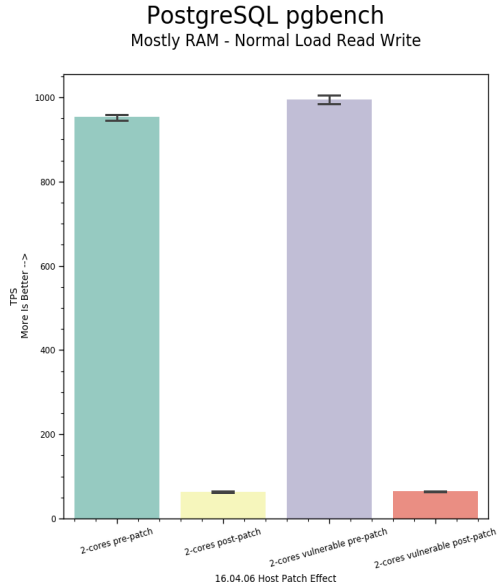



(b) 16 Cores - Read Write

**Figure 5.8: Host Patch: Mostly RAM - Heavy Contention Results**

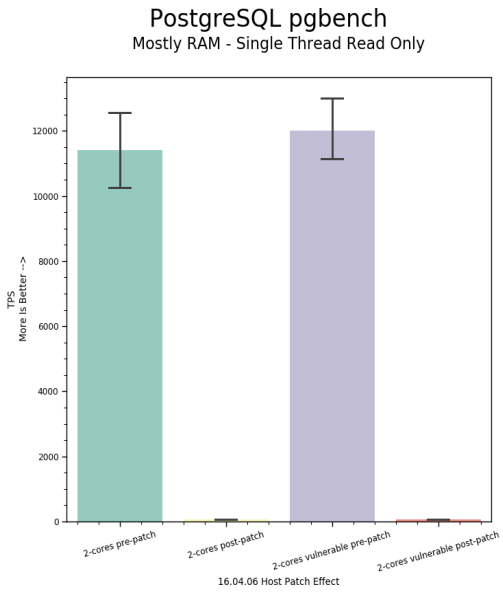



(a) 16 Cores - Read Only

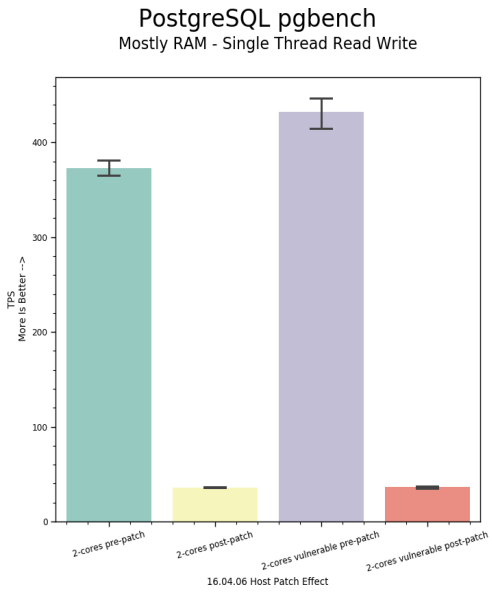



(b) 16 Cores - Read Write

**Figure 5.9: Host Patch: Mostly RAM - Normal Load Results**




(a) 16 Cores - Read Only




(b) 16 Cores - Read Write

**Figure 5.10: Host Patch: Mostly RAM - Single Thread Results**

**Table 5.6: On-Disk Tests Benchmark Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Heavy Contention - RO | 16.04.03 | **3 cores vulnerable** | 18.04.01 | 3 cores |
| Heavy Contention - RW | 16.04.01 | **3 cores vulnerable** | 18.04.01 | 3 cores vulnerable |
| Normal Load - RO | 16.04.03 | **3 cores vulnerable** | 18.04.01 | 3 cores |
| Normal Load - RW | 16.04.04 | **3 cores vulnerable** | 18.04.01 | 3 cores vulnerable |
| Single Thread - RO | 16.04.02 | **1 core vulnerable** | 18.04.01 | 3 cores vulnerable |
| Single Thread - RW | 16.04.05 | 1 core vulnerable | **18.04.01** | 2 cores |

### 5.2.4 On-Disk

Table 5.6 shows not only are all of the Ubuntu 16.04.06 tests better with mitigations disabled, they are also almost always the best. Figures A.19, A.20, A.21, A.22, A.23, and A.24 display a large discrepancy between Ubuntu 18.04.01 and Ubuntu 18.04.02. Interestingly this is the same discrepancy shown in the Mostly RAM tests,

**Table 5.7: On-Disk Tests Benchmark Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Heavy Contention - RO | 10.05 | 96.73 | 199.28 | 16.27 |
| Heavy Contention - RW | 5.48 | 94.23 | 169.88 | 33.72 |
| Normal Load - RO | 7.07 | 97.56 | 199.50 | 40.49 |
| Normal Load - RW | 3.88 | 89.75 | 175.03 | 26.73 |
| Single Thread - RO | 19.40 | 67.51 | 198.99 | 17.57 |
| Single Thread - RW | 14.90 | 11.14 | 165.15 | 8.35 |

but in the opposite direction. While most of the figures show the same stair-stepping visualization as other tests where having more cores is critical, several of the Ubuntu 18.04.02 core results are instead more randomized, likely due to the impact of the mitigations.

**Table 5.8: DaCapo Benchmark Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Eclipse | 16.04.01 | 3 cores vulnerable | 18.04.01 | **3 cores** |
| H2 | **16.04.01** | 2 cores vulnerable | 18.04.01 | 2 cores vulnerable |
| Jython | 16.04.02 | 3 cores vulnerable | 18.04.01 | **3 cores** |
| Tradebeans | 16.04.02 | 3 cores vulnerable | 18.04.02 | **3 cores** |
| Tradesoap | 16.04.02 | 3 cores vulnerable | 18.04.01 | **3 cores vulnerable** |

**Table 5.9: DaCapo Benchmark Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Eclipse | 7.18 | 61.24 | 1.90 | 63.02 |
| H2 | 32.53 | 34.19 | 3.13 | 19.37 |
| Jython | 4.42 | 72.65 | 2.23 | 68.41 |
| Tradebeans | 4.52 | 76.93 | 0.18 | 76.16 |
| Tradesoap | 12.75 | 86.21 | 2.35 | 87.69 |

## 5.3 DaCapo

Table 5.9 shows that all of the DaCapo benchmarks are affected by core count, as expected. Some benchmarks, Eclipse, H2, and Jython, do poorly with one core but the performance with two and three cores is similar. The other DaCapo benchmarks, Tradebeans and Tradesoap, are more sensitive to having two than three cores. The table additionally shows that within patch levels there is little percent difference. H2 has a significant variance when looking at Ubuntu 16 patches, but Figure A.26 shows that is due to an unusually large amount of variance within 16.04.04. Table 5.8 makes it seem like disabling the mitigations is the right choice for better performance on the DaCapo benchmarks. However these graphs show that the gain is not significant for any of the benchmarks.

**Table 5.10: SciMark: ANSI C Benchmark Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Dense LU | 16.04.05 | **1 core** | 18.04.01 | 1 core |
| FFT | **16.04.02** | 3 cores vulnerable | 18.04.01 | 2 cores |
| Jacobi SOR | 16.04.05 | 1 core | 18.04.01 | **1 core** |
| Monte Carlo | 16.04.05 | **1 core** | 18.04.01 | 1 core |
| Sparse MM | 16.04.05 | **1 core** | 18.04.01 | 1 core |
| Composite | 16.04.05 | **1 core** | 18.04.01 | 1 core |

## 5.4 SciMark

Tables 5.10 and 5.12 show that there is little difference between either patch levels or core number for either ANSI C or Java SciMark. This is likely because these are fairly small math benchmarks that both processor and operating system creators use to optimize their product before it is released. It is still useful to investigate them

**Table 5.11: SciMark: ANSI C Benchmark Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Dense LU | 1.84 | 2.39 | 1.65 | 3.36 |
| FFT | 6.75 | 1.97 | 1.08 | 2.30 |
| Jacobi SOR | 1.29 | 2.36 | 1.00 | 2.53 |
| Monte Carlo | 2.18 | 3.29 | 1.19 | 3.94 |
| Sparse MM | 1.22 | 1.93 | 1.77 | 3.96 |
| Composite | 1.22 | 1.67 | 1.37 | 3.07 |

**Table 5.12: SciMark: Java Benchmark Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Dense LU | 16.04.01 | 3 cores vulnerable | 18.04.01 | **1 core** |
| FFT | 16.04.05 | 2 cores vulnerable | **18.04.02** | 2 cores |
| Jacobi SOR | 16.04.05 | **2 cores vulnerable** | 18.04.01 | 1 core |
| Monte Carlo | 16.04.05 | **2 cores vulnerable** | 18.04.01 | 1 core |
| Sparse MM | 16.04.05 | 1 core | **18.04.01** | 1 core |
| Composite | 16.04.05 | 2 cores vulnerable | 18.04.01 | **1 core** |

**Table 5.13: SciMark: Java Benchmark Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Dense LU | 2.59 | 1.10 | 0.95 | 2.31 |
| FFT | 3.59 | 2.37 | 1.03 | 2.18 |
| Jacobi SOR | 5.98 | 1.92 | 1.58 | 2.09 |
| Monte Carlo | 5.41 | 1.45 | 1.24 | 2.23 |
| Sparse MM | 5.70 | 1.52 | 1.44 | 2.11 |
| Composite | 3.78 | 1.42 | 0.93 | 1.69 |

to ensure nothing has gone catastrophically wrong, considering how many scientific simulations use thousands of these operations, but their lack of a meaningful winning category for any given SciMark test is both unsurprising and comforting.

## 5.5 Encoding

**Table 5.14: Encoding Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| LAME MP3 | 16.04.05 | 2-cores vulnerable | **18.04.01** | 1 core |
| x264 | 16.04.02 | 3 cores vulnerable | 18.04.01 | **3 cores** |

**Table 5.15: Encoding Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| LAME MP3 | 2.71 | 2.89 | 1.28 | 1.93 |
| x264 | 2.14 | 81.67 | 1.61 | 84.35 |

Figure A.43 and Table 5.14 both show that the video encoding benchmark, x264, behaves as expected, with more cores doing better than fewer cores. Additionally, table 5.15 shows patch level having a negligible impact. This benchmark is another interesting potential case where the host patch may have affected the Ubuntu 18 vulnerable versus default tests. While this behaves as expected, the audio encoding benchmark, LAME MP3, does not. Table 5.15 shows that LAME MP3 is not affected by either core number or patch level, and while A.42 shows some differences, they are negligible. This is likely due to high levels of optimizations on LAME to enable it to run well on anything.

**Table 5.16: Ray Tracing Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| C-Ray | 16.04.05 | 3 cores vulnerable | 18.04.01 | **3 cores** |
| Sunflow | 16.04.02 | 3 cores vulnerable | 18.04.01 | **3 cores** |

**Table 5.17: Ray Tracing Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| C-Ray | 1.20 | 98.44 | 0.70 | 97.54 |
| Sunflow | 7.08 | 35.32 | 4.57 | 35.28 |

## 5.6 Ray Tracing

The ray tracing benchmarks both perform as expected. In table 5.16 it can be seen that both ray tracing programs perform similarly and are able to utilize a larger number of cores successfully. Additionally it shows that both perform better in Ubuntu 18 than Ubuntu 16 as they are likely optimized for 64-bit operating systems. This is supported by there being little variance in patch levels according to table 5.17. Figure A.45 shows that the Sunflow Rendering System benchmark will not run on less than two cores. Interestingly, while both benchmarks perform better without mitigations for 16.04.06, it is the opposite for 18.04.02, where they are faster even than 16.04.06 without mitigations. Taking a close look at Figures A.45 and A.44 indicates that it is the 64-bit optimizations rather than the patch to the host system which is responsible.

**Table 5.18: Compression Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| 7-Zip | 16.04.02 | 3 cores vulnerable | **18.04.01** | 2 cores |
| Gzip | 16.04.05 | 3 cores vulnerable | **18.04.01** | 3 cores |

**Table 5.19: Compression Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| 7-zip | 3.96 | 64.36 | 0.09 | 66.61 |
| Gzip | 1.14 | 5.20 | 3.90 | 6.33 |

## 5.7 Compression

The compression tests perform as expected. It is interesting to see that for both tests Ubuntu 18.04.01 was the best; see table 5.18. The 7-Zip benchmark has a slightly unusual bimodal distribution, Figure A.46, and seem to be able to use two cores at most, while the Gzip tests do not display the same distribution, Figure A.47. 7-Zip's bimodal distribution is likely caused by caching; the test is compressing a file three times, the first time will always be the worst and the next test runs will better because the data will be in one of the cache levels. If this test was improved it would compress the file multiple times each run and average the time, or compress a file that was much larger than the last level cache. Interestingly, both figure A.46 and table 5.18 show that for 7-Zip, there are no benefits of more than two cores, while gzip, Figure A.47, has little difference at any core level. Additionally, table 5.19 show that there is little variance with different patch levels for either test. Lastly, the gzip Ubuntu 18 tests have a better average speed than their Ubuntu 16 counterparts, meaning that the `tar` function in Ubuntu 18 is better optimized.

## 5.8 Miscellaneous

The tests in this section show a typical reaction in that the test runs with higher core numbers were almost universally the best.

**Table 5.20: Miscellaneous Best Averages**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Bork | 16.04.06 | 3 cores vulnerable | **18.04.01** | 2 cores |
| MAFFT | 16.04.01 | 3 cores vulnerable | 18.04.02 | **3 cores vulnerable** |
| R | 16.04.02 | 2 cores vulnerable | 18.04.02 | **3 cores vulnerable** |
| SQLite | **16.04.04** | 1 core | 18.04.02 | 2 cores |
| Uengine - Fullscreen | 16.04.04 | **3 cores vulnerable** | 18.04.01 | 3 cores |
| Uengine - Windowed | 16.04.04 | **3 cores** | 18.04.01 | 3 cores |

**Table 5.21: Miscellaneous Percent Difference of Average**

| Benchmark | 16 Patch | 16.04.02 Core | 18 Patch | 18.04.02 Core |
|---|---|---|---|---|
| Bork | 38.62 | 39.53 | 1.43 | 57.32 |
| MAFFT | 8.42 | 87.75 | 0.10 | 84.84 |
| R | 5.28 | 4.45 | 0.38 | 3.00 |
| SQLite | 41.23 | 33.77 | 3.33 | 25.56 |
| Uengine - Fullscreen | 3.99 | 94.56 | 0.75 | 93.64 |
| Uengine - Windowed | 4.21 | 94.87 | 1.61 | 94.03 |

### 5.8.1 Bork

The Bork test is unusual as an outlier in this regard, but this is likely caused by its larger variance when compared to the other tests in this section. This variance can be seen in Figure A.48, where some of the plots stretch in unusual ways. Additionally, when comparing the percent difference in the testing runs, seen in Table 5.21, the Ubuntu 18 Patches have smallest percent difference, confirming how similar the data is between the two runs. In contrast the different core levels of 18 have the largest variance, but this is likely due the extreme anomalous behavior shown in 2-cores vulnerable. Figure A.48 shows that Ubuntu 18 tests are faster on average than their

Ubuntu 16 counterparts, and table 5.20 has Ubuntu 18.04.01 as the best result.

### 5.8.2 MAFFT

The tests for MAAFT behaved as expected. Figure A.49 shows that the number of cores provided was the biggest impact, and when compared Ubuntu 18 always was faster than Ubuntu 16 for the same number of cores or vulnerabilities, which likely means that it is optimized for 64-bit operating systems. Table 5.21 shows a small variance between patches, and figure A.49 likewise indicates a small variance between vulnerable and default core configurations.

### 5.8.3 R

On the graphs of the R benchmark, Figure A.50, there appears to be a large amount of variance. This isn't unusual because as a timing benchmark with such a small footprint (less than two seconds), it is likely that this is due to background processes on either the virtual box or host system. The benchmark apears to be affected by the mitigations given that its best averages are without them, as shown in Table 5.20. Table 5.21 shows that the percent difference between runs is small, so it seems affected by the mitigations, but without more data from other sources it is hard to know if it is just a coincidence.

### 5.8.4 SQLite

The SQLite tests are unusual in that they do not perform better for having more cores, easily seen in table 5.20. This is likely because, as a Lite model, it is designed without the heavy threading that would take advantage of the higher core number.

Additionally, these tests have a high degree of variance, table 5.21, which is likely caused by some of the tests having extreme outliers; see figure A.51.

### 5.8.5 Unigine

Both tests for Unigine behaved as expected, with the number of cores being the most important aspect. Figures A.53 and A.52 show that the number of cores provided was the biggest impact. When compared Ubuntu 16 always was faster than Ubuntu 18, which is likely due to the Unigine Sanctuary test being the oldest Unigine test and thus optimized for the 32-bit operating system. Although table 5.20 shows a difference between the windowed and fullscreen tests for mitigation impact, looking at figures A.53 and A.52 show that the data is so close it is negligible. Additionally, table 5.21 shows that the data between patches has little variance.

This chapter presented both an analysis of and the results of the experiment. The next chapter contains information on potential future work, including future experiments and additional variables.

# Chapter 6

# FUTURE WORK

Recording the total impact from the mitigations to stop Meltdown and Spectre is a Herculean task. The number of potential combinations is enormous, and the computation time for the tests inside this thesis is non-trivial. Considering how quickly new Speculative Execution attacks are being discovered, it is very likely that a new attack will be discovered and mitigated before the testing is complete.

## 6.1 Motherboards

Additional experimentation is needed to determine if the motherboard, specifically the chipsets on the motherboard, are affected by the mitigations to the speculative execution attacks. The impact that the patch to the Host machine had in this experiment shows patches which should have no impact, due to the processor not being of the type being patched, can still have an impact. Although investigation into the Host machine's motherboard indicated a lack of the chip being patched, further searching revealed that, to the OS, the motherboard is only known as "legacy," which could indicate a blanket patching strategy to deal with the unknown chipset.

## 6.2 Combination

To have a comprehensive analysis, all possible combinations of mounting, operating system, hardware, and active mitigation must be tested extensively. Table 6.1 shows an example of how many variables could exist in a full experiment.

**Table 6.1: Further Testing Variables for Combination**

| Mount | OS | Hardware | Mitigation |
|---|---|---|---|
| Bare Metal | Mac OS | ARM Cortex R Series | L1TF |
| Oracle Virtual Box | Android | ARM Cortex A Series | Spec. Store Bypass |
| VMware Workstation | Windows 7 | AMD Ryzen | Spectre v1 |
| VMware Fusion | Windows 8 | AMD Athlon | Spectre v2 |
| Parallels Desktop | Windows 10 | AMD Threadripper | Meltdown |
| | Ubuntu 12.04 | Intel Nehalem | |
| | Ubuntu 14.04 | Intel Sandy Bridge | |
| | Ubuntu 16.04 | Intel Ivy Bridge | |
| | Ubuntu 18.04 | Intel Haswell | |
| | | Intel Broadwell | |
| | | Intel Skylake | |
| | | Intel Kaby Lake | |
| | | Intel Coffee Lake | |
| | | Intel Cannon Lake | |
| | | Intel Ice Lake | |

### 6.2.1 Mounting

Because the mitigations affect Virtual Machines differently than a Bare Metal mount, both should be tested extensively. However, there will be slight differences in how the internal guest machines are affected due to the implementation that each Virtual Machine software takes. Therefore, multiple Virtual Machine softwares should be used, in order to normalize out implementation specific details.

### 6.2.2 Operating Systems

This work addressed only two operating systems (Linux 16.04 and Linux 18.04) and their respective patch levels on a virtual machine operating on a Windows 10 host. To see the full impact on an operating system level, all affected operating systems should be benchmarked. Additionally, because the mitigations affect the performance of virtual machines, a fully comprehensive study must take into account all possible permutations of Virtual Machine Software, Client OS, and Host OS, as well as the respective patch-levels of each.

### 6.2.3 Hardware

Due to the number of CPUs affected by the mitigations, it is equally prudent to see how much each CPU has potentially lost in performance. The tests should be performed across different micro-architectures by company, but also different variations within each architectural generation should be tested. These variations can vary greatly from each in core number, cache amount, memory hierarchy, and clock rate.

#### 6.2.3.1 Mitigation

In order to see the full impact of each mitigation tests must be performed with different mitigations turned on or off. Unless that is done the impact of the individual mitigations and how they perform with each other mitigation will only be guesswork. However, that means that for each new speculative execution attack and subsequent mitigation that the amount of testing to do increases massively. Additionally, new speculative execution attacks are being published frequently enough that a new one can be published in the middle of experimentation. This happened to this paper, as the MDS attacks and their software mitigation were published after these results were gathered.

## 6.3 Computation Time

The benchmarks demonstrated in this thesis take 20 hours to complete, for a single run. Therefore, the time to test all of the potential combinations represented in Table 6.1 is:

$$tt * (os * hw * m + vm * (hos * m) * (gos * m) * hw)$$

The section for baremetal testing time and the section for virtual machine testing time are separated, and the variables correspond as follows: test time: tt; hardware permutations: hw; mitigation permutations: m; virtual machine software permutations: vm; host os permutations: hos; and guest os permutations: gos. The result of this equation is:

$$20 * (5 * 15 * 32 + 4 * (5 * 32) * (5 * 32) * 15) = 30768000 \text{ hours}$$

which is equivalent to about 3512 years of computation time. This doesn't take into account that not all operating systems can run virtual machines, or that there would

be setup time in order to decompile the kernel in order to test some mitigations like Spectre v1. This equation also does not take into account all of the different patch levels that should be checkd for both Host and Guest machines. Additionally, if further research into the patch on the host machine shows that motherboard chipsets are affected by the patches, another variable and all of it's permutations are added to the equation. While the tests obviously shouldn't be done sequentially, the total computing times shows how large this task is.

This chapter covered potential future experiments, both to find additional quantitative results and to find or eliminate additional possible variables. The next and final chapter presents the conclusions of the paper.

# Chapter 7

# CONCLUSION

The result of the tests were not surprising. As expected, almost all benchmarks benefited greatly from an increased core count. These benchmarks were either the newer or more complex benchmarks whose developers optimized for the additional performance the increase in core number can give. Simpler benchmarks were less effected by either core count or having mitigations turned off. Surprisingly, even though Ubuntu 16 did not have the Meltdown mitigation, kpti, enabled by default, it did not outperform Ubuntu 18. This shows that being 64-bit native is more of a performance impact than the kpti mitigation. The biggest surprise was the impact of the accidental patch to the host, which according to Windows, should have only affected VIA-based systems. Although many benchmarks showed affect, the degree of change was mostly small.

These tests showed that currently, the number of benchmarks that performed better without mitigations was almost equal to the number that performed better with the mitigations enabled. Thus, while many benchmarks are affected by the software mitigations, it is not the overwhelming majority. Therefore, users seeking performance gain must research their individual usecase, as there does not seem to be a blanket gain by disabling mitigations.

Some microarchitectural exploits can only be fixed in hardware. Processor manufacturers are attempting to create new designs that are not vulnerable to Meltdown- and Spectre-type exploits However, even as these designs are created new exploits are discovered that need to be protected against, causing additional temporary software


**Figure 7.1: Branchless Doom**

mitigations. Consumers seeking top performance will need to continue to follow the news about these exploits and mitigations.

The problem of performance versus security will continue as long as computers exist, if not longer. Figure 7.1 shows the video-game Doom reprogrammed entirely in move instructions, thus removing all branch instructions and becoming immune to Meltdown- and Spectre-type attacks. Unfortunately, as it can only render one frame every seven hours, it shows that this isn't a feasible strategy for protecting against these attacks [28].

The flaws exposed by Meltdown and Spectre are pebbles in an avalanche, causing security researchers to tear apart hardware diagrams and undocumented optimizations. Eventually they will be fixed, and new optimizations will be added that will in turn be targeted, starting the cycle anew.

BIBLIOGRAPHY


[1] 7-cpu. Intel haswell.

[2] 7-Zip. 7-zip website, 2019.

[3] A. S. E. K. W. R. A. K. Andrea Mambretti, Matthias Neugschwandtner. Let's not speculate: Discovering and analyzing speculative execution attacks. *IBM Research*, 2018.

[4] ASUS. Z97-a.

[5] O. Benchmarking. Bork file encrypter, 2016.

[6] O. Benchmarking. 7-zip compression, 2018.

[7] O. Benchmarking. C-ray, 2018.

[8] O. Benchmarking. Dacapo benchmark, 2018.

[9] O. Benchmarking. Gzip compression, 2018.

[10] O. Benchmarking. Java scimark, 2018.

[11] O. Benchmarking. R benchmark, 2018.

[12] O. Benchmarking. Sanctuary demo, 2018.

[13] O. Benchmarking. Scimark, 2018.

[14] O. Benchmarking. Sqlite, 2018.

[15] O. Benchmarking. Sunflow rendering system, 2018.

[16] O. Benchmarking. Timed mafft alignment, 2018.

[17] O. Benchmarking. x264, 2018.

[18] O. Benchmarking. Lame mp3 encoding, 2019.

[19] S. Bhunia and M. Tehranipoor. Chapter 16 - system level attacks & countermeasures. In S. Bhunia and M. Tehranipoor, editors, *Hardware Security*, pages 419 – 448. Morgan Kaufmann, 2019.

[20] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[21] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, ANU, 2006. http://www.dacapobench.org.

[22] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. *CoRR*, abs/1811.05441, 2018.

[23] N. Cburnett, Inductiveload. 4 stage pipeline svg, 2015.

[24] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. 02 2018.

[25] J. Corbet. Kaiser: hiding the kernel from user space. 2017.

[26] J. Corbet. Meltdown strikes back: the l1 terminal fault vulnerability. 2018.

[27] C. Cunningham. What is hyper-threading and simultaneous multithreading?, 2017.

[28] domas.

[29] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 693–707, New York, NY, USA, 2018. ACM.

[30] R. D. Finn, J. Clements, and S. R. Eddy. Hmmer web server: interactive sequence similarity searching. *Nucleic acids research*, 39(suppl_2):W29–W37, 2011.

[31] A. Fog. *Instruction tables*. Technical University of Denmark, 2018.

[32] R. Fraile. How gzip compression works. 2014.

[33] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 955–972, 2018.

[34] P. Grosjean and S. Steinhaus. R benchmarks, 2008.

[35] T. P. G. D. Group. Postgresql.

[36] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[37] J. P. Huelsenbeck and F. Ronquist. Mrbayes: Bayesian inference of phylogenetic trees. *Bioinformatics*, 17(8):754–755, 2001.

[38] Intel. Intel analysis of speculative execution side channels, 2018.

[39] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. Spoiler: Speculative load hazards boost rowhammer and cache attacks. *CoRR*, abs/1903.00446, 2019.

[40] T. Jain and T. Agrawal. The haswell microarchitecture-4th generation processor.

[41] K. Katoh, K. Misawa, K.-i. Kuma, and T. Miyata. Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic acids research*, 30(14):3059–3066, 2002.

[42] T. kernel development community. L1tf - l1 terminal fault.

[43] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[44] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[45] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[46] H. Li, K.-S. Leung, and M.-H. Wong. idock: A multithreaded virtual screening tool for flexible ligand docking. In *2012 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, pages 77–84. IEEE, 2012.

[47] 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[48] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[49] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[50] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom. Fallout: Reading kernel writes from user space. 2019.

[51] V. Organization. x264.

[52] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[53] R. Pozo and B. R. Miller. Scimark 2.0.

[54] T. L. Project. The lame project.

[55] P. Rademacher. Ray tracing: graphics for the masses.

[56] J. Sanders. Why mds vulnerabilities present a threat as serious as spectre and meltdown, 2019.

[57] M. Schwarz, C. Canella, L. Giner, and D. Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. 2019.

[58] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv:1905.05726*, 2019.

[59] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.

[60] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[61] A. L. Shimpi. Intel's sandy bridge architecture exposed. 2010.

[62] SQLite. Sqlite tutorial, 2018.

[63] J. Stecklina and T. Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018.

[64] M. Stevens, P. Karpman, and T. Peyrin. Freestart collision for full sha-1. Cryptology ePrint Archive, Report 2015/967, 2015. `https://eprint.iacr.org/2015/967`.

[65] V. Strobel. Pold87/academic-keyword-occurrence: First release, Apr. 2018.

[66] G. Torres. How the cache memory works. 2007.

[67] P. Tracy. Zombieload attacks may affect all intel cpus since 2011: What to do now, 2019.

[68] Unigine. Sanctuary demo, 2007.

[69] Unigine. About us, 2019.

[70] 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 Security Symposium*. USENIX Association, August 2018.

[71] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

[72] S. van Schaik, A. Milburn, S. sterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[73] M. Vanhoef and F. Piessens. All your biases belong to us: Breaking rc4 in wpa-tkip and TLS. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 97–112, Washington, D.C., 2015. USENIX Association.

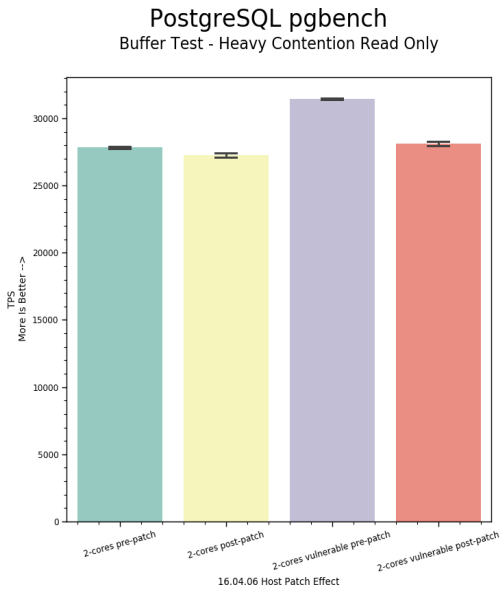
[74] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [70].

[75] 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)*, pages 719–732, 2014.

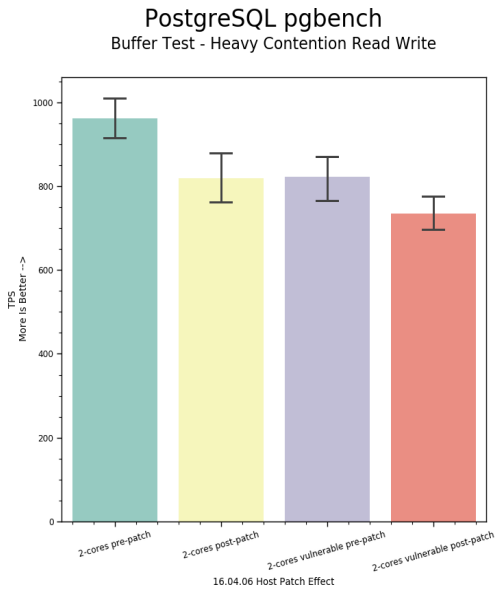
# APPENDICES

## Appendix A

## INDIVIDUAL TEST RESULT GRAPHS

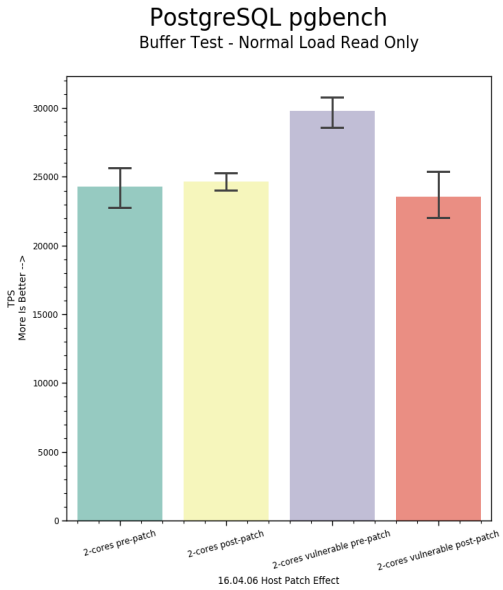



(a) 16 Cores - Read Only

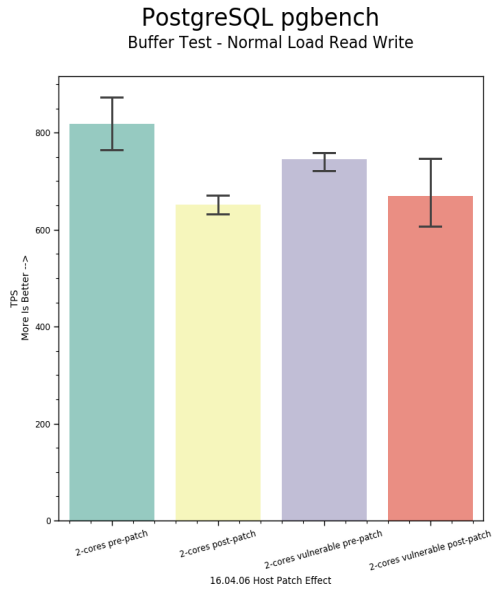



(b) 16 Cores - Read Write

**Figure A.1: Host Patch: pgbench Buffer Test - Heavy Contention Results**

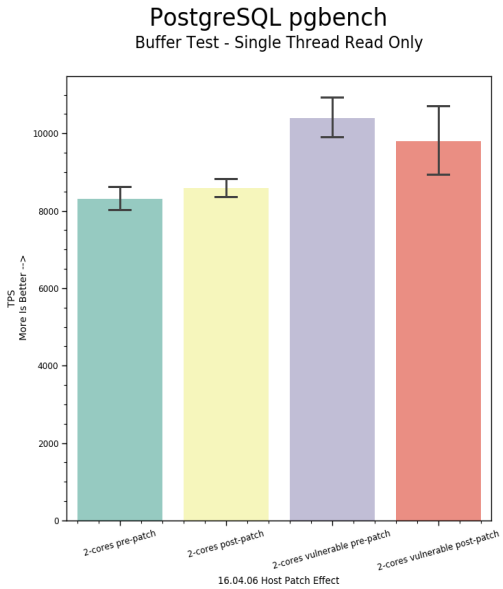



(a) 16 Cores - Read Only

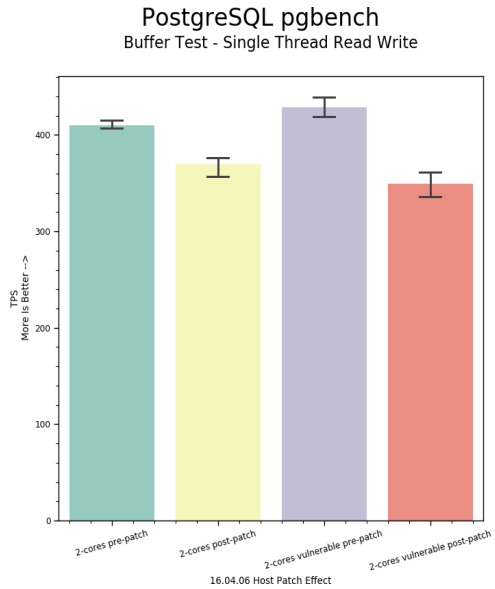



(b) 16 Cores - Read Write

**Figure A.2: Host Patch: pgbench Buffer Test - Normal Load Results**

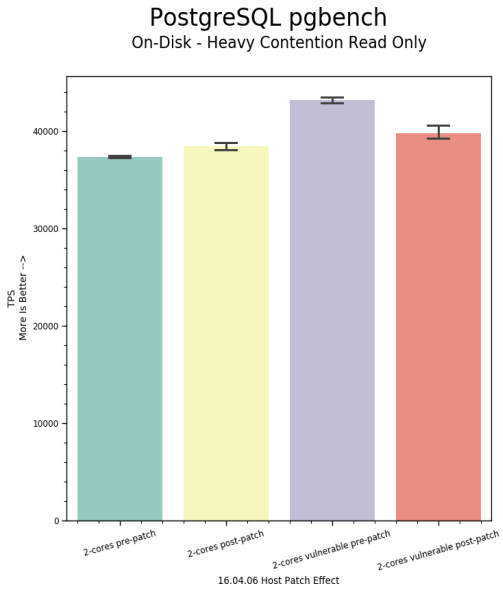



(a) 16 Cores - Read Only

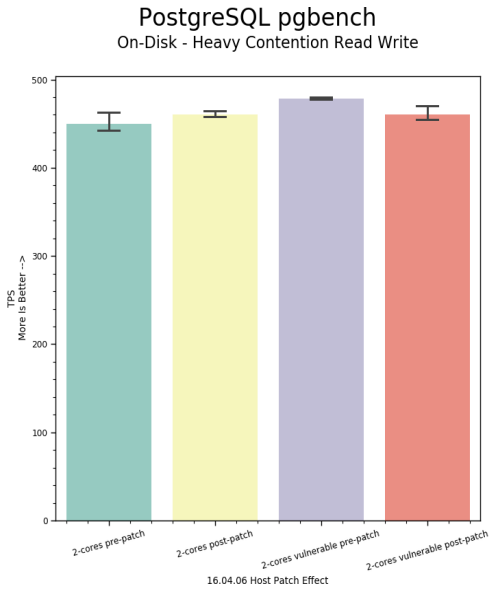



(b) 16 Cores - Read Write

**Figure A.3: Host Patch: pgbench Buffer Test - Single Thread Results**

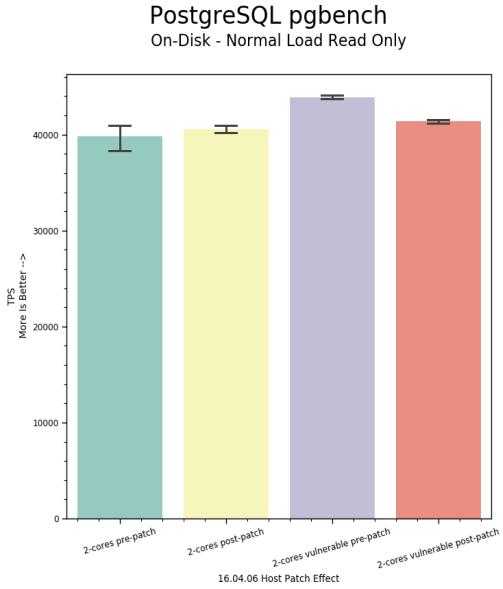



(a) 16 Cores - Read Only

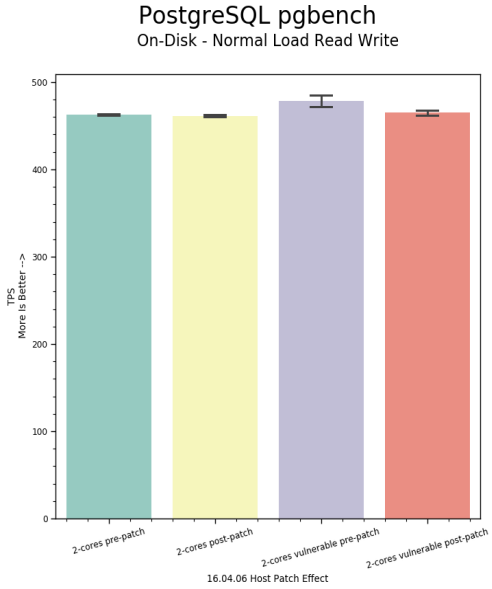



(b) 16 Cores - Read Write

**Figure A.4: Host Patch: pgbench On-Disk - Heavy Contention Results**

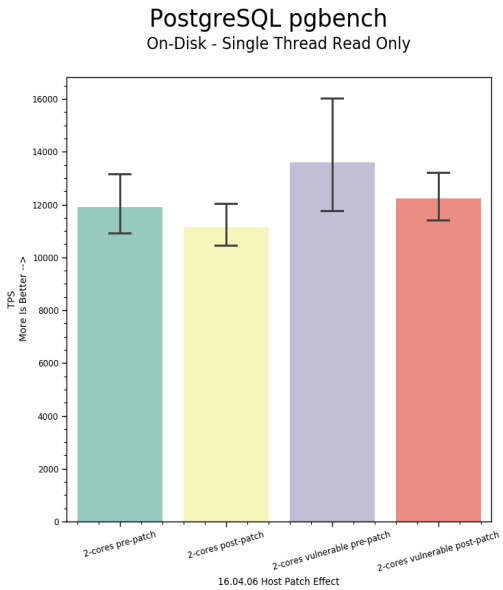



(a) 16 Cores - Read Only

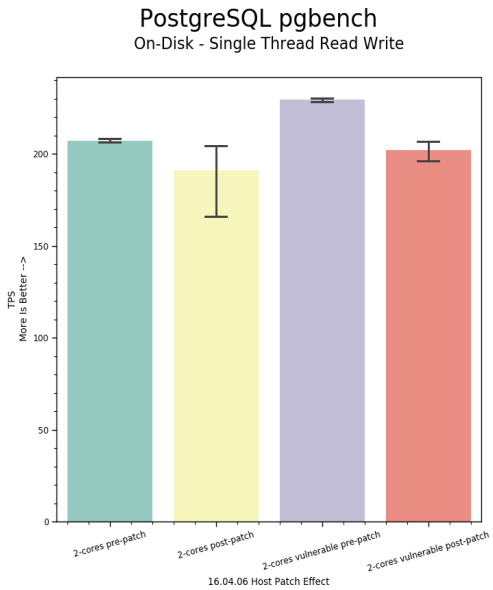



(b) 16 Cores - Read Write

**Figure A.5: Host Patch: pgbench On-Disk - Normal Load Results**

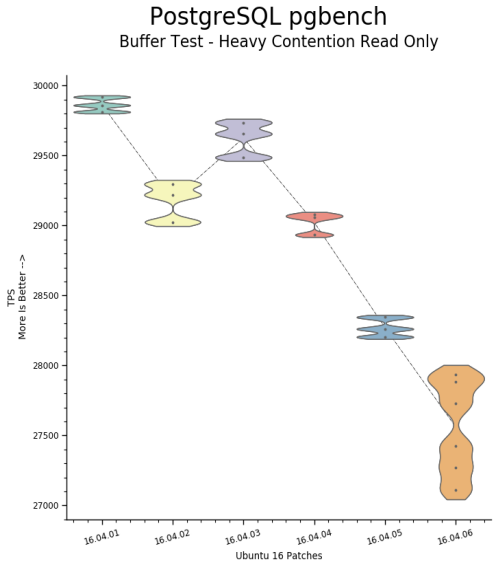



(a) 16 Cores - Read Only

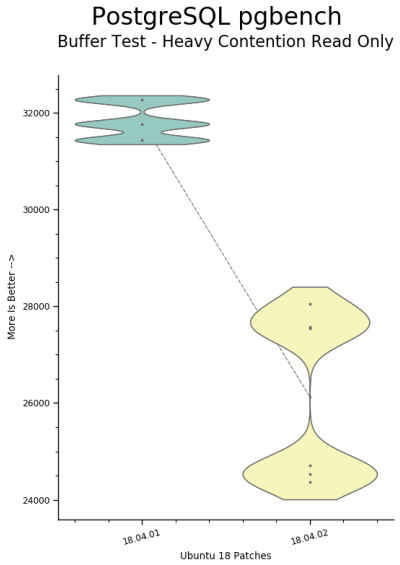



(b) 16 Cores - Read Write

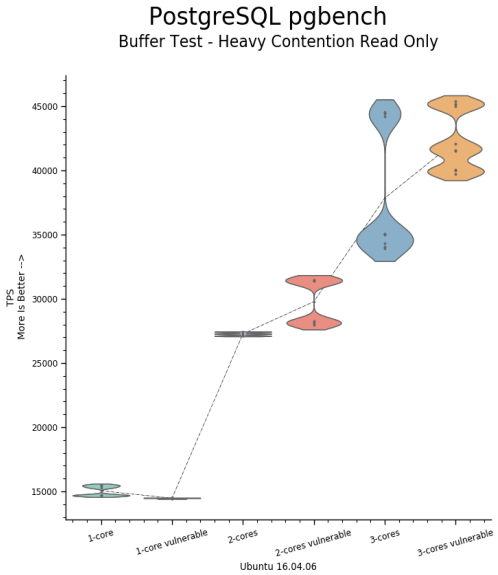
**Figure A.6: Host Patch: pgbench On-Disk - Single Thread Results**



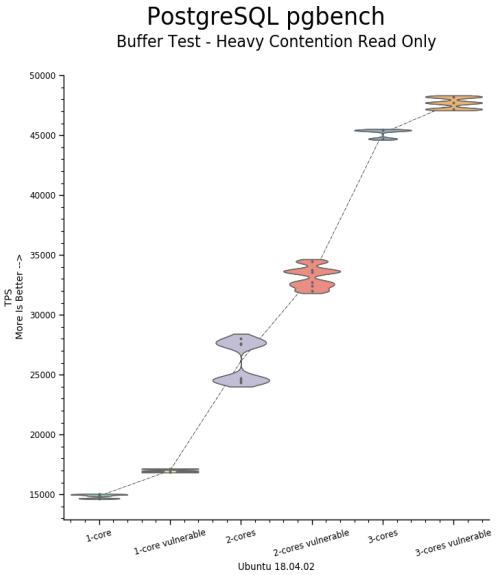

(a) 16 Patches




(b) 18 Patches

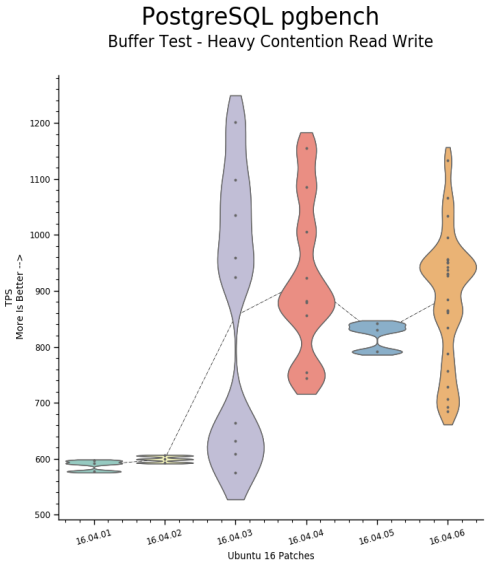



(c) Ubuntu 16.04.06 Cores

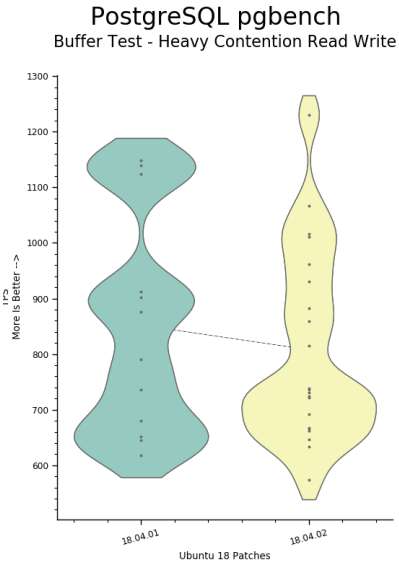



(d) Ubuntu 18.04.2 Cores

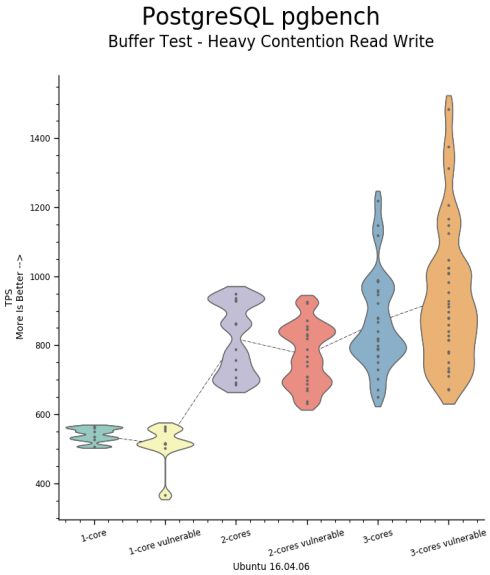
**Figure A.7: pgbench Buffer Test - Heavy Contention - Read Only Results**



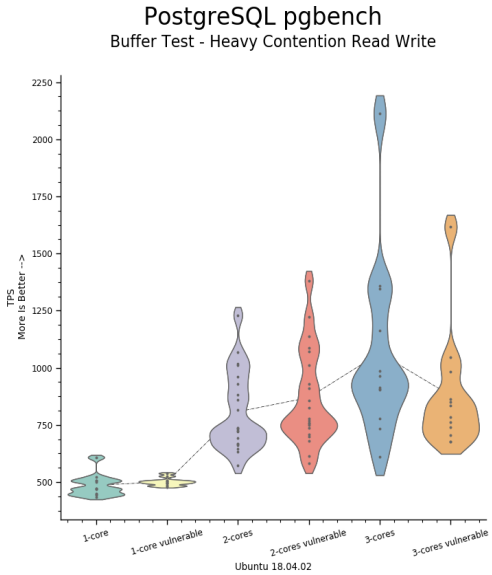

(a) 16 Patches




(b) 18 Patches

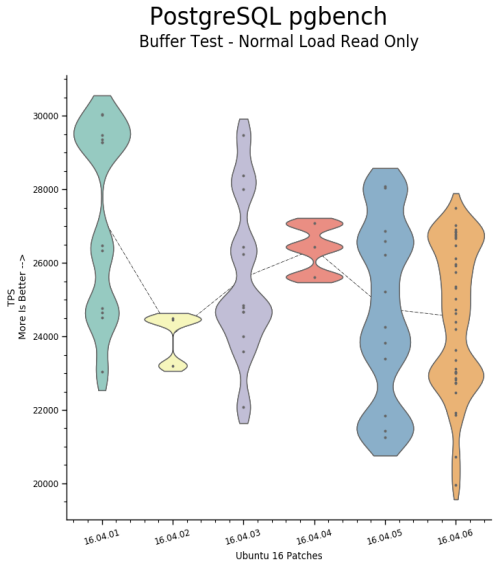



(c) Ubuntu 16.04.06 Cores

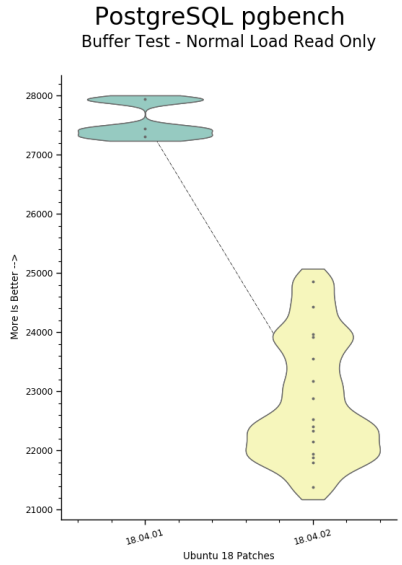



(d) Ubuntu 18.04.2 Cores

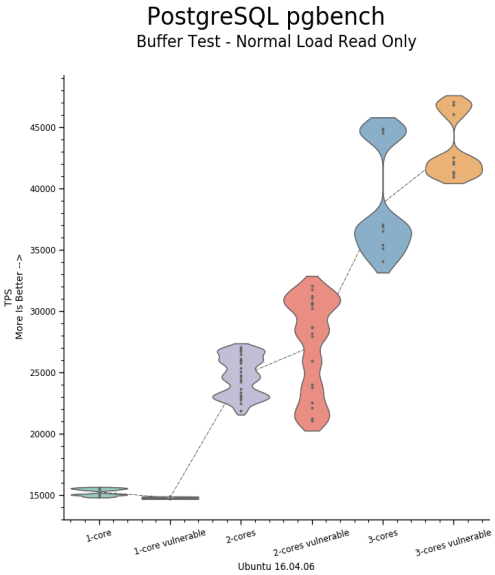
**Figure A.8: pgbench Buffer Test - Heavy Contention - Read Write Results**



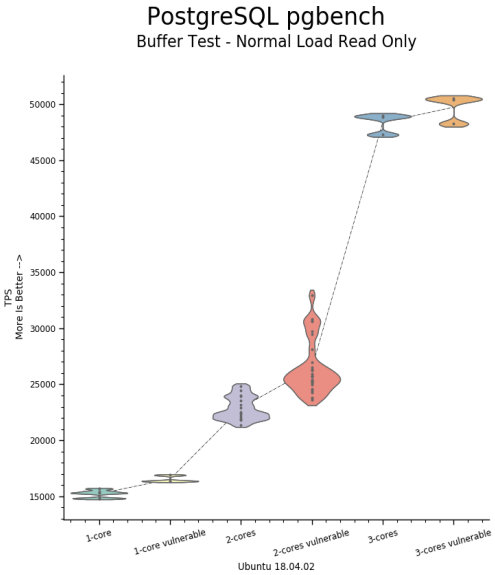

(a) 16 Patches




(b) 18 Patches

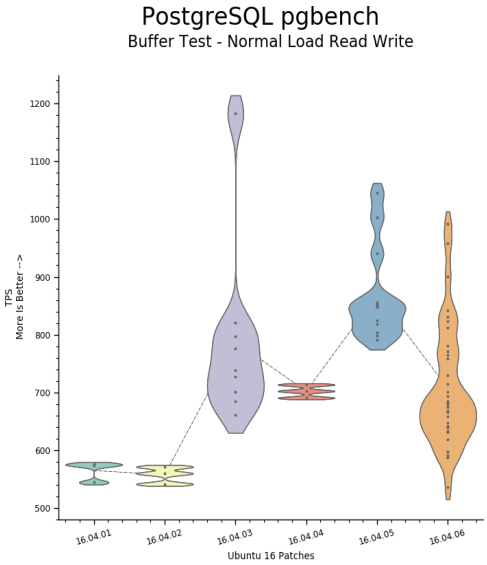



(c) Ubuntu 16.04.06 Cores

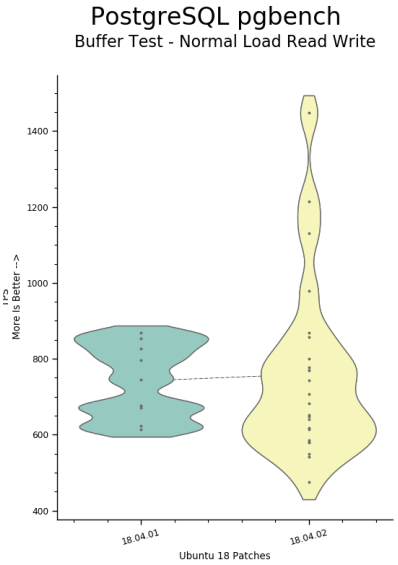



(d) Ubuntu 18.04.2 Cores

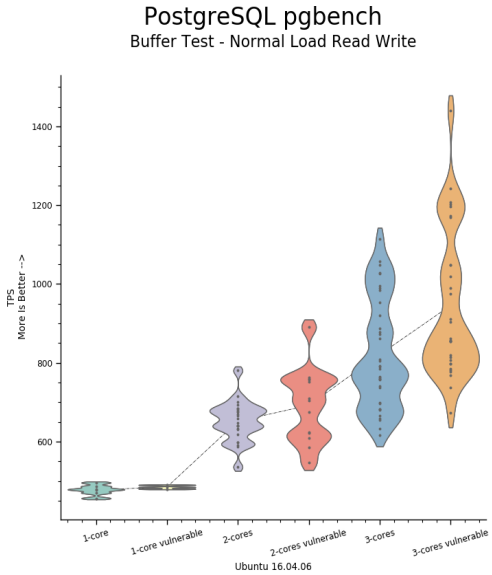
**Figure A.9: pgbench Buffer Test - Normal Load - Read Only Results**



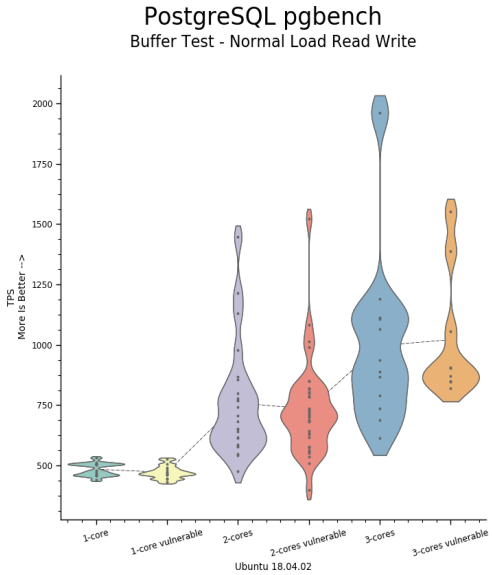

(a) 16 Patches




(b) 18 Patches

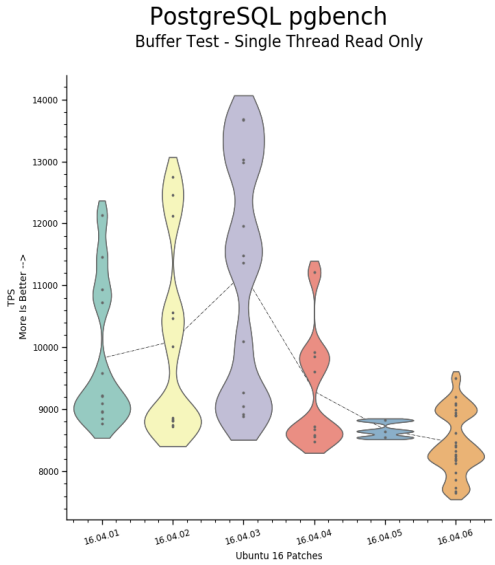



(c) Ubuntu 16.04.06 Cores

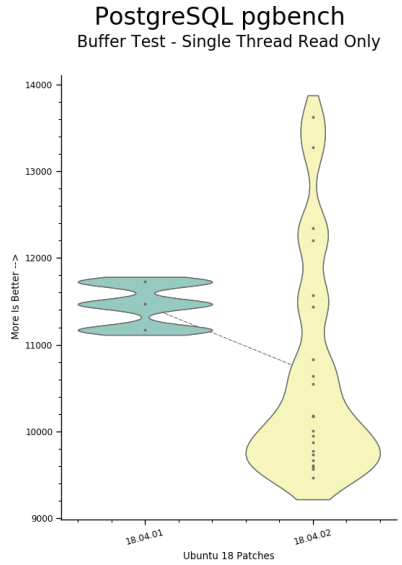



(d) Ubuntu 18.04.2 Cores

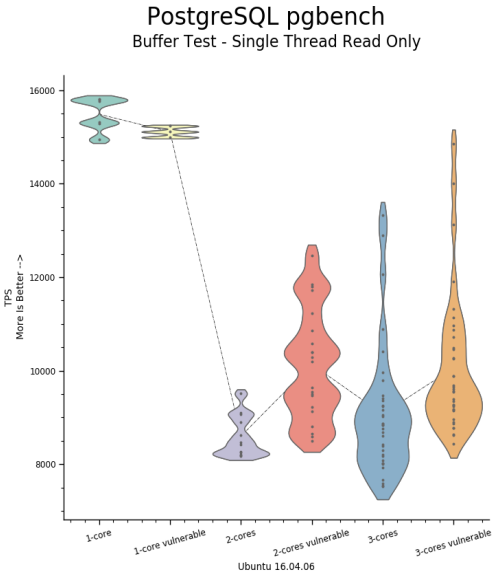
**Figure A.10: pgbench Buffer Test - Normal Load - Read Write Results**



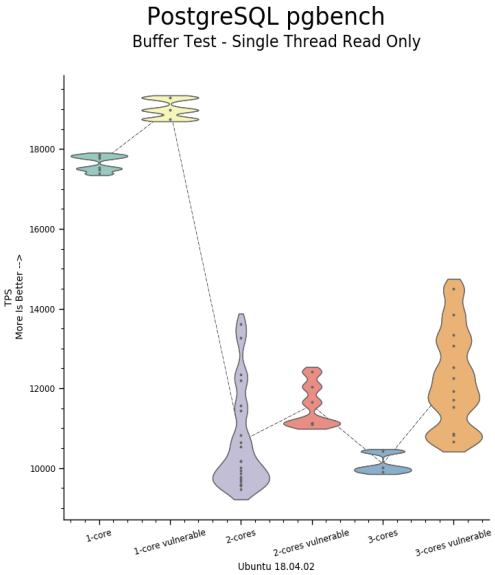

(a) 16 Patches




(b) 18 Patches

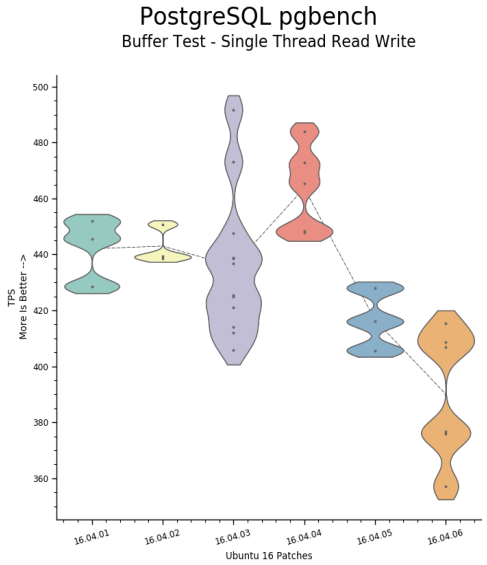



(c) Ubuntu 16.04.06 Cores

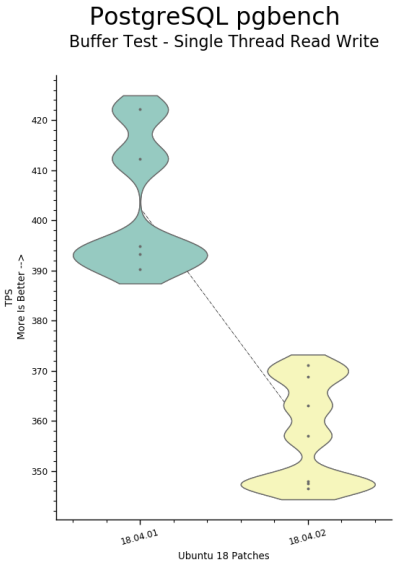



(d) Ubuntu 18.04.2 Cores

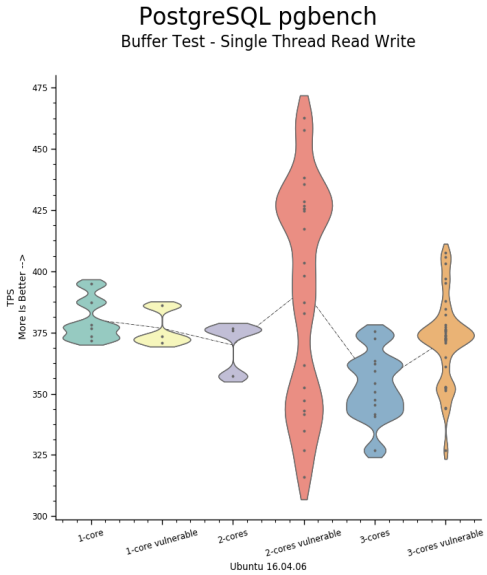
**Figure A.11: pgbench Buffer Test - Single Thread - Read Only Results**



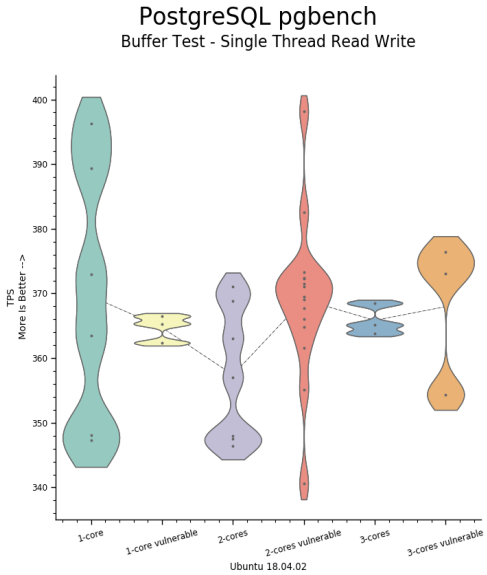

(a) 16 Patches




(b) 18 Patches

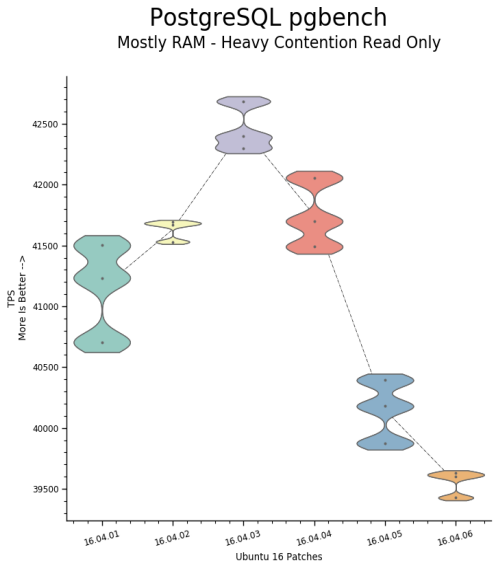



(c) Ubuntu 16.04.06 Cores

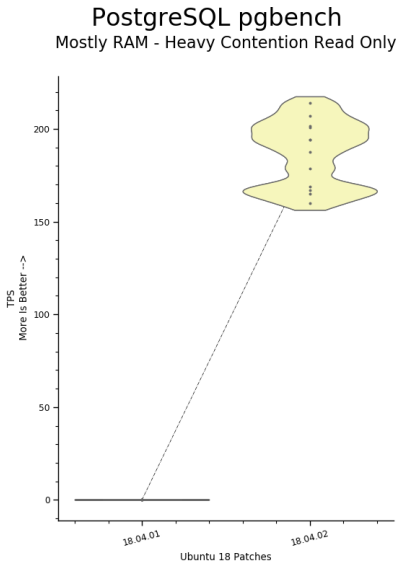



(d) Ubuntu 18.04.2 Cores

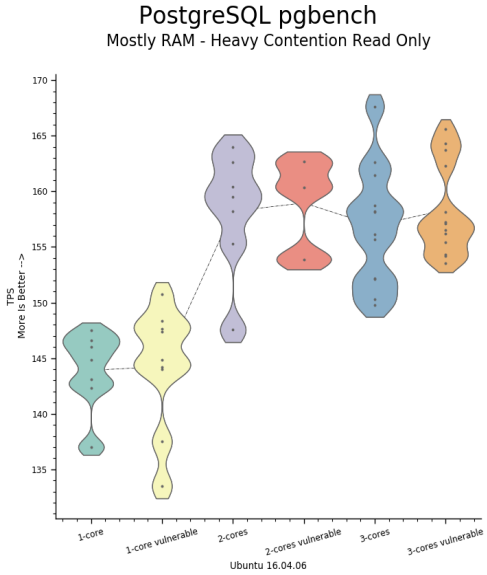
**Figure A.12: pgbench Buffer Test - Single Thread - Read Write Results**



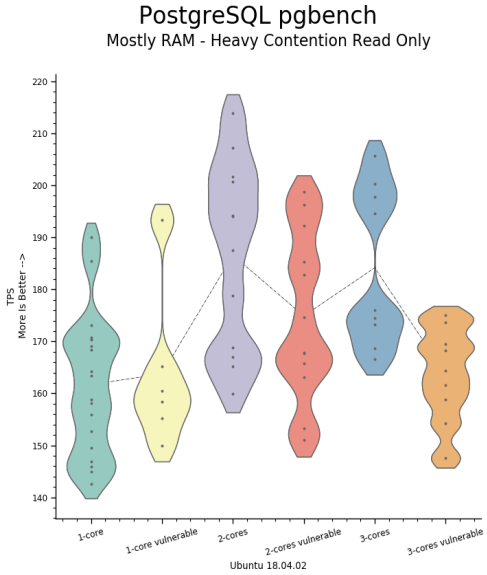

(a) 16 Patches




(b) 18 Patches

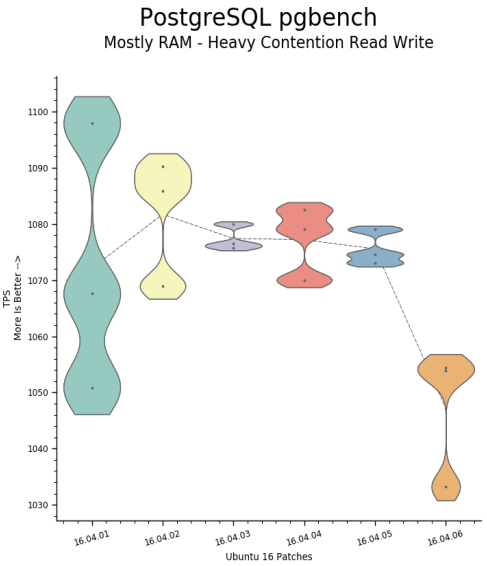



(c) Ubuntu 16.04.06 Cores

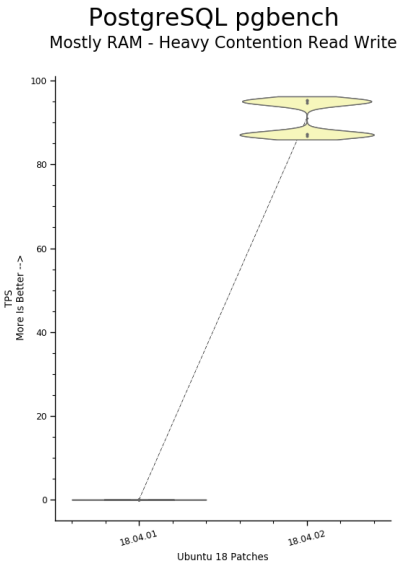



(d) Ubuntu 18.04.2 Cores

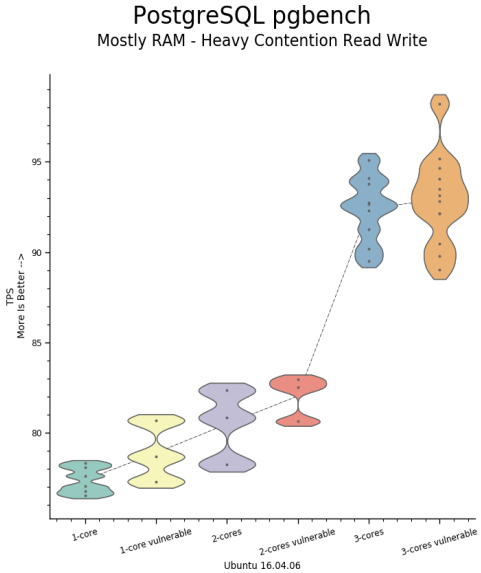
**Figure A.13: pgbench Mostly RAM - Heavy Contention - Read Only Results**



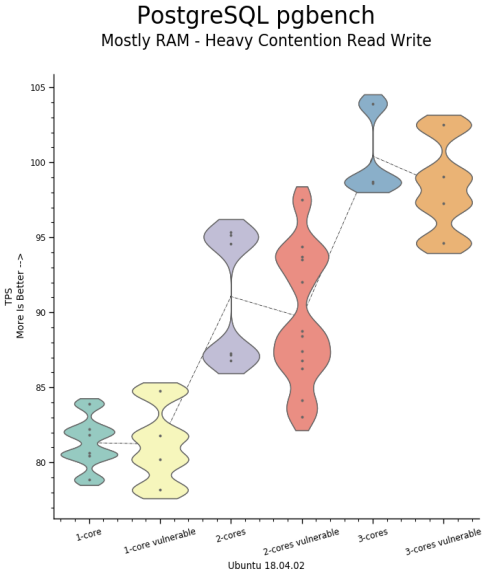

(a) 16 Patches




(b) 18 Patches

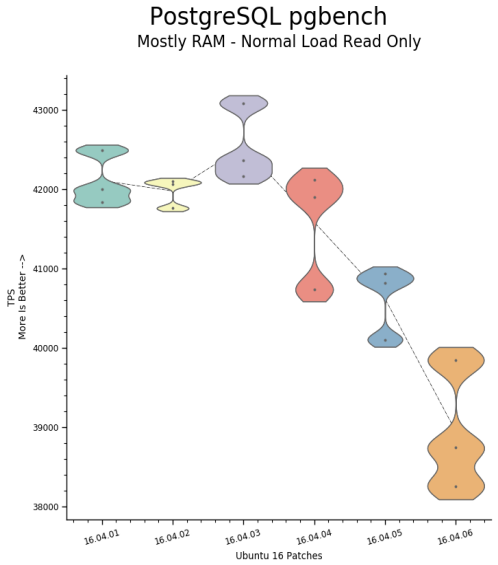



(c) Ubuntu 16.04.06 Cores

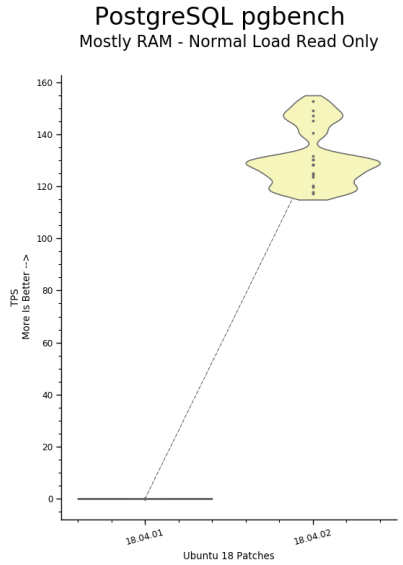



(d) Ubuntu 18.04.2 Cores

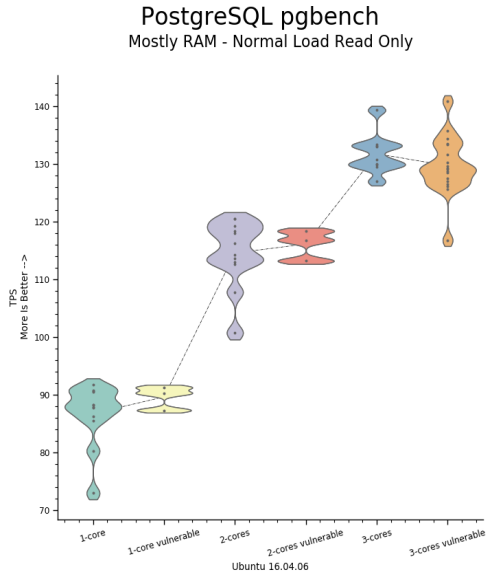
**Figure A.14: pgbench Mostly RAM - Heavy Contention - Read Write Results**



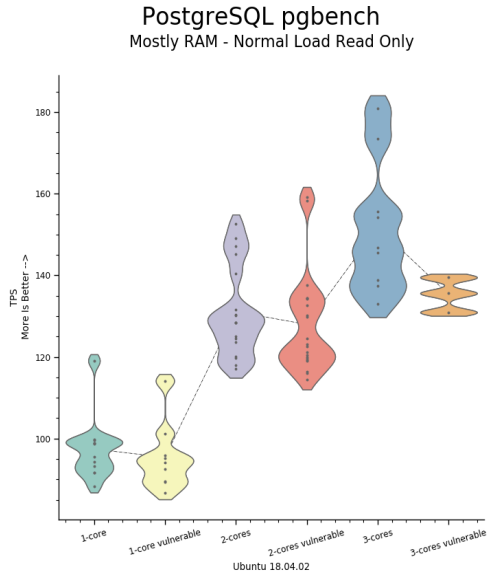

(a) 16 Patches




(b) 18 Patches

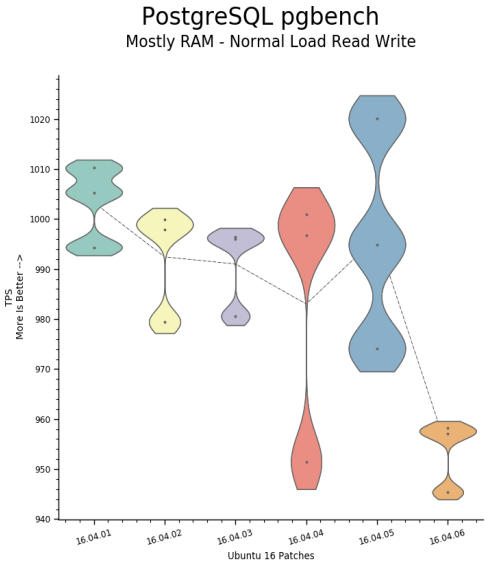



(c) Ubuntu 16.04.06 Cores

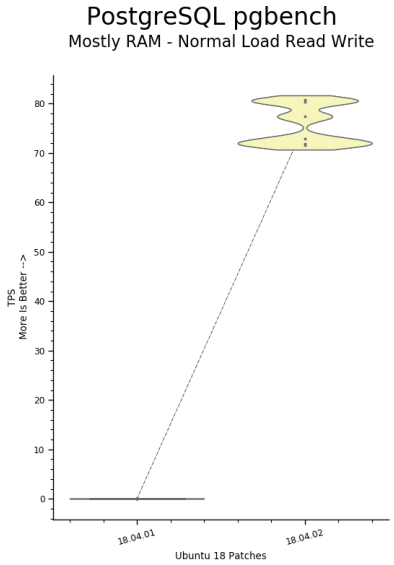



(d) Ubuntu 18.04.2 Cores

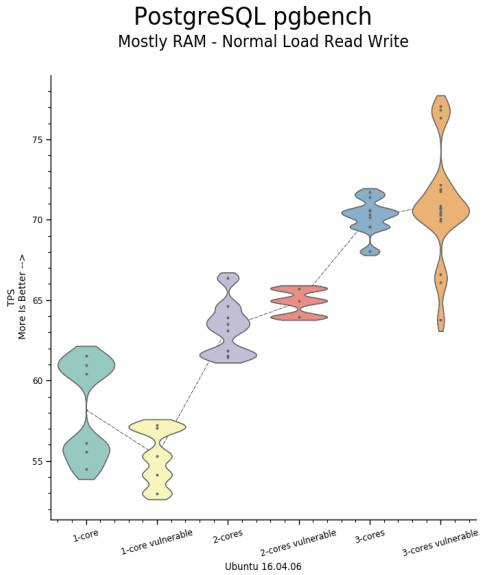
**Figure A.15: pgbench Mostly RAM - Normal Load - Read Only Results**



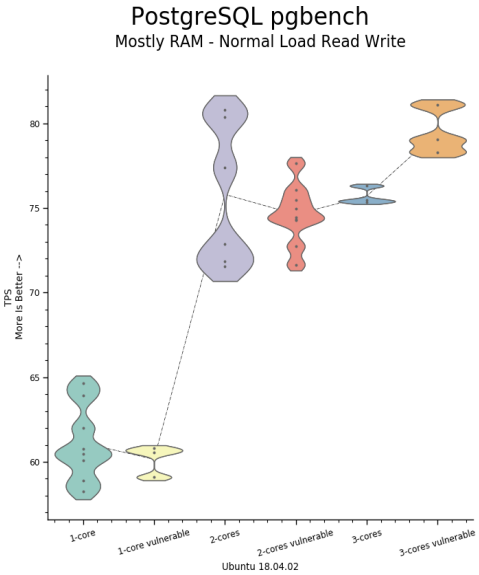

(a) 16 Patches




(b) 18 Patches

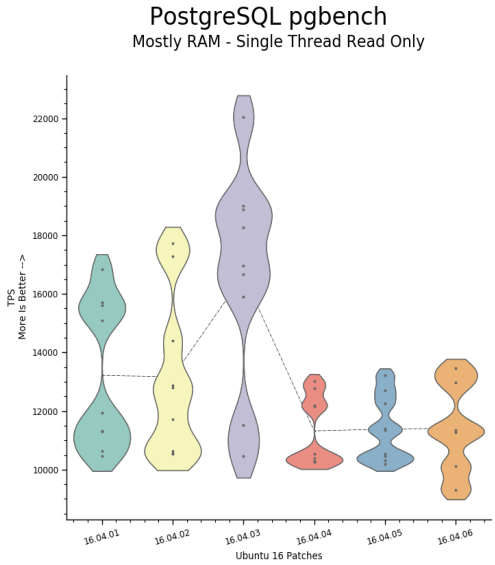



(c) Ubuntu 16.04.06 Cores

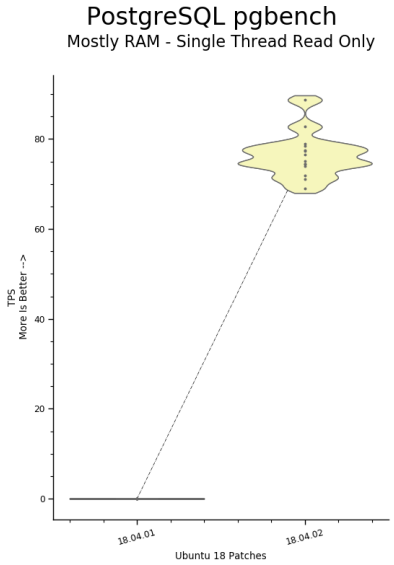



(d) Ubuntu 18.04.2 Cores

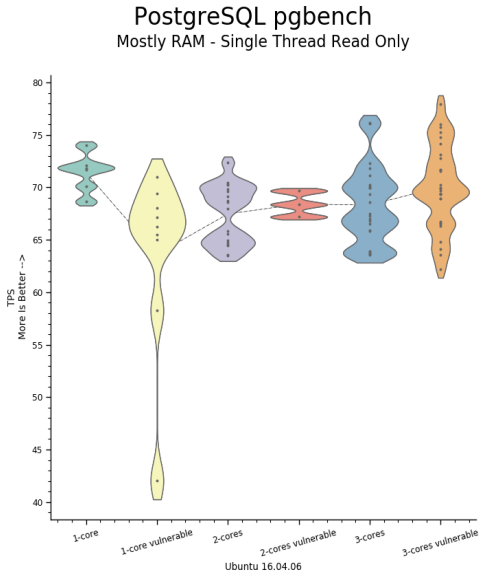
**Figure A.16: pgbench Mostly RAM - Normal Load - Read Write Results**



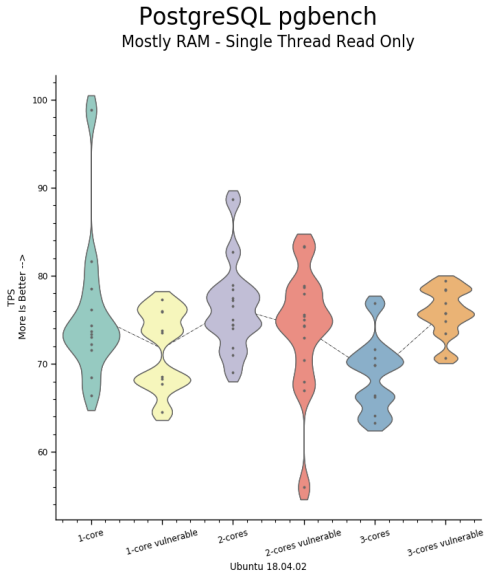

(a) 16 Patches




(b) 18 Patches

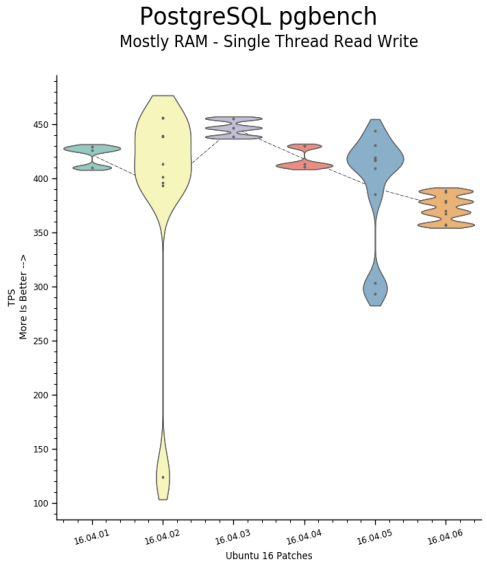



(c) Ubuntu 16.04.06 Cores

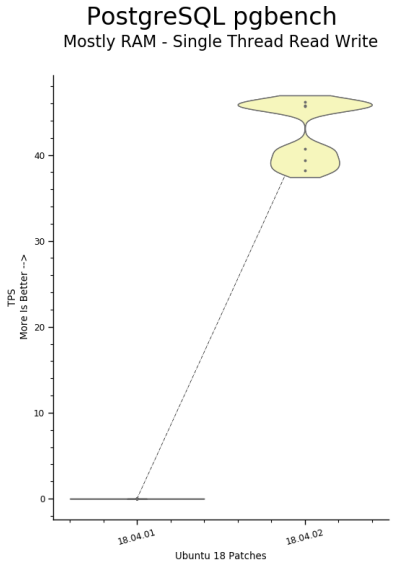



(d) Ubuntu 18.04.2 Cores

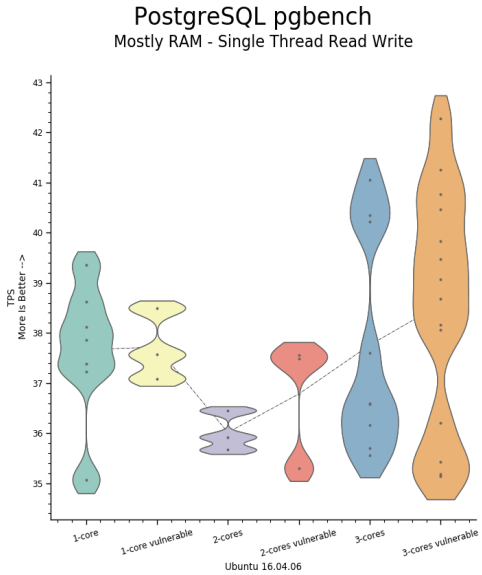
**Figure A.17: pgbench Mostly RAM - Single Thread - Read Only Results**



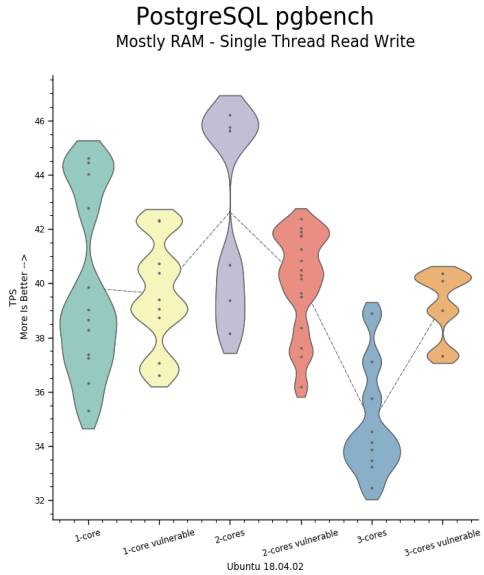

(a) 16 Patches




(b) 18 Patches

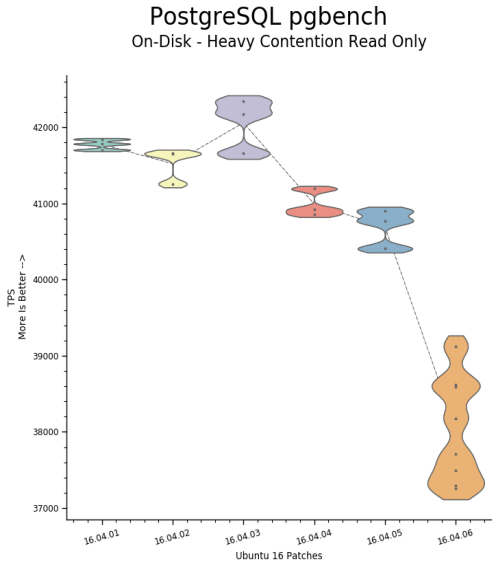



(c) Ubuntu 16.04.06 Cores

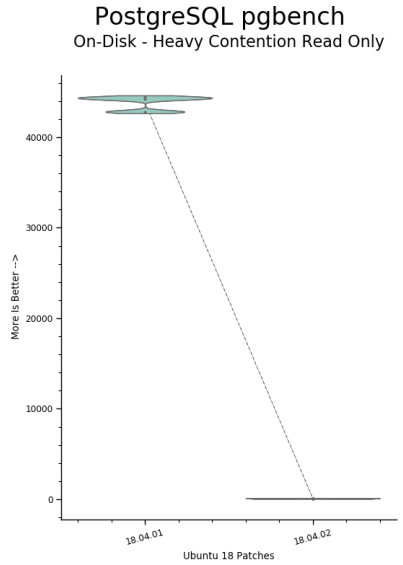



(d) Ubuntu 18.04.2 Cores

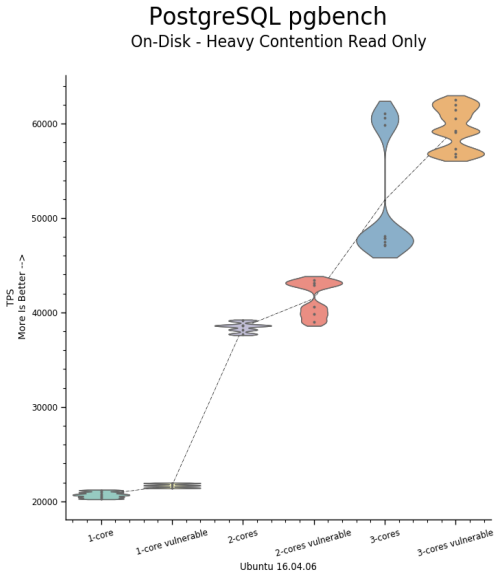
**Figure A.18: pgbench Mostly RAM - Single Thread - Read Write Results**



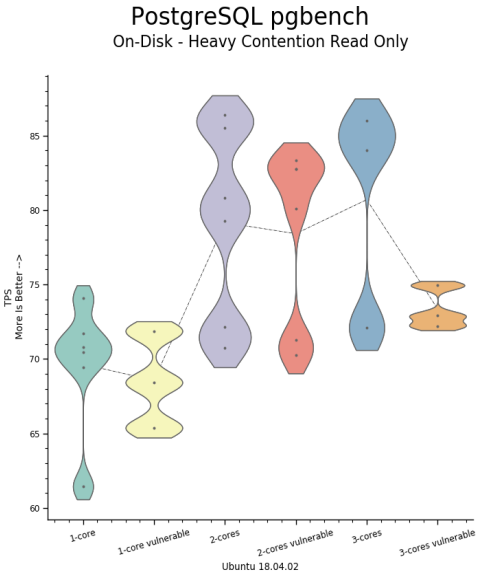

(a) 16 Patches




(b) 18 Patches

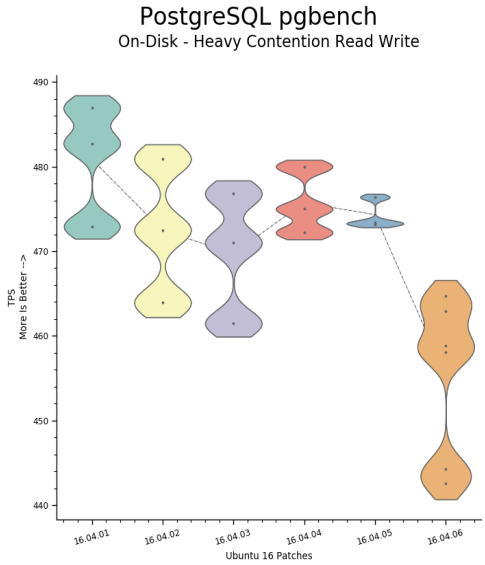



(c) Ubuntu 16.04.06 Cores

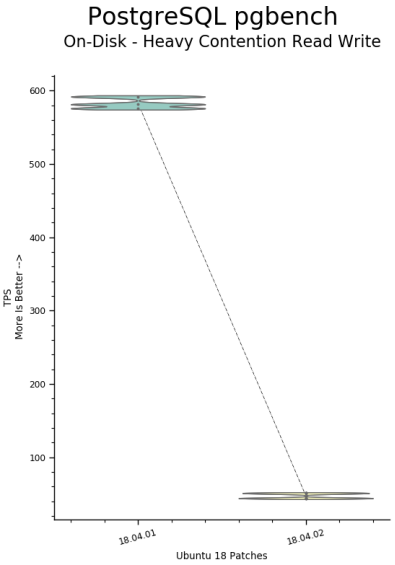



(d) Ubuntu 18.04.2 Cores

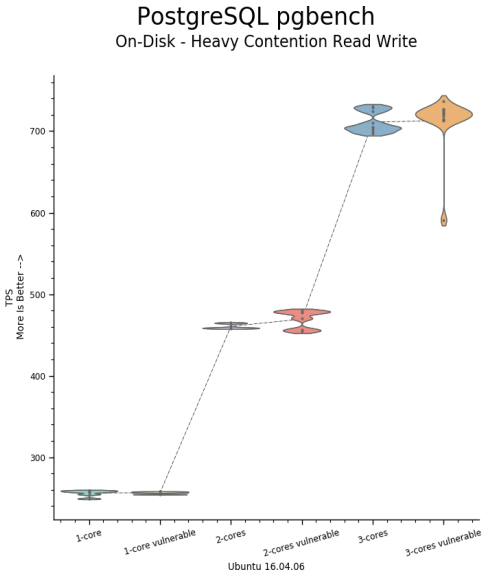
**Figure A.19: pgbench On-Disk - Heavy Contention - Read Only Results**



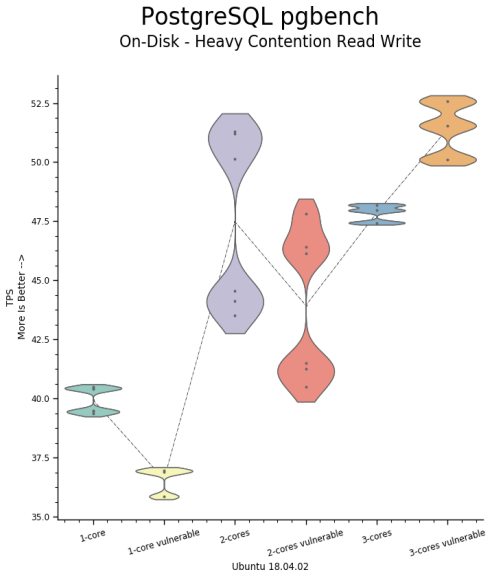

(a) 16 Patches




(b) 18 Patches

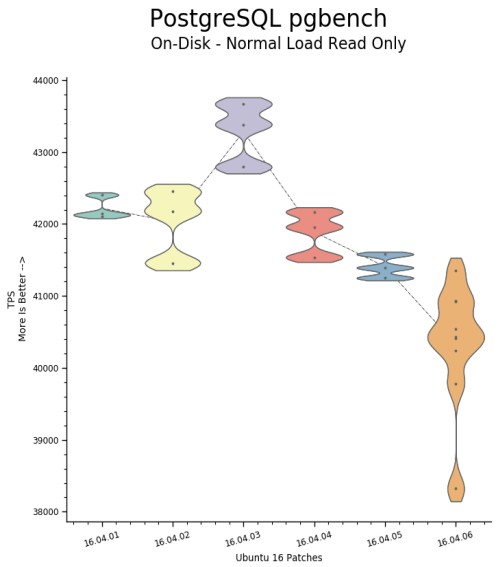



(c) Ubuntu 16.04.06 Cores

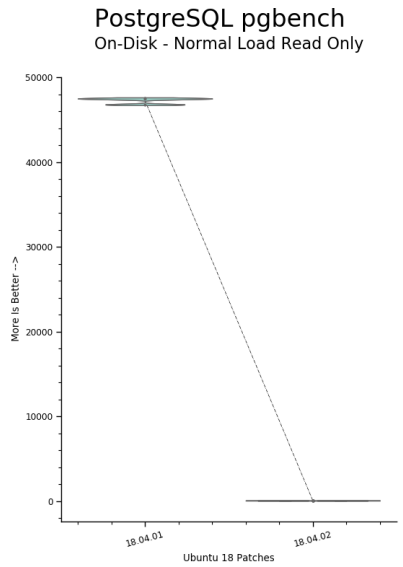



(d) Ubuntu 18.04.2 Cores

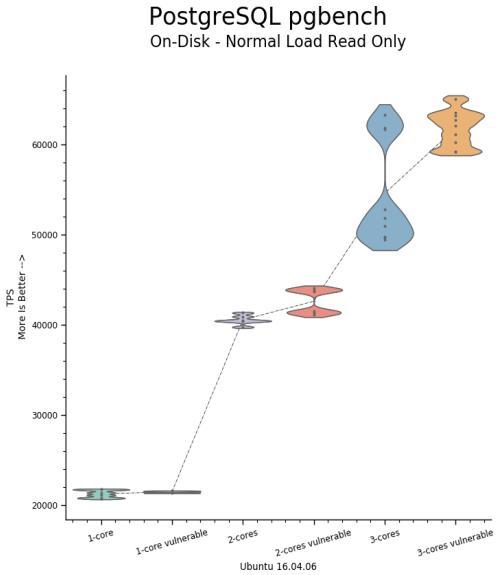
**Figure A.20: pgbench On-Disk - Heavy Contention - Read Write Results**



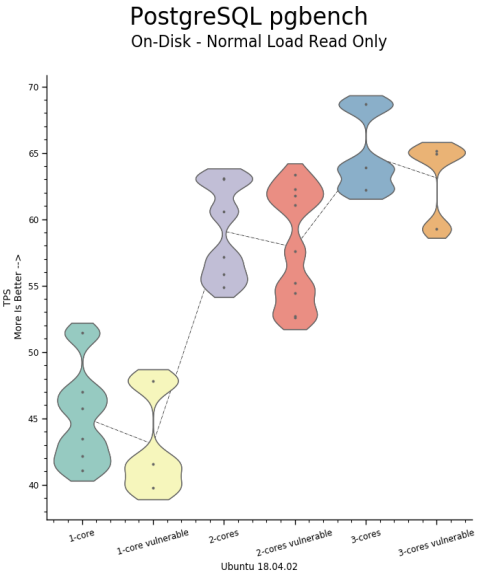

(a) 16 Patches




(b) 18 Patches

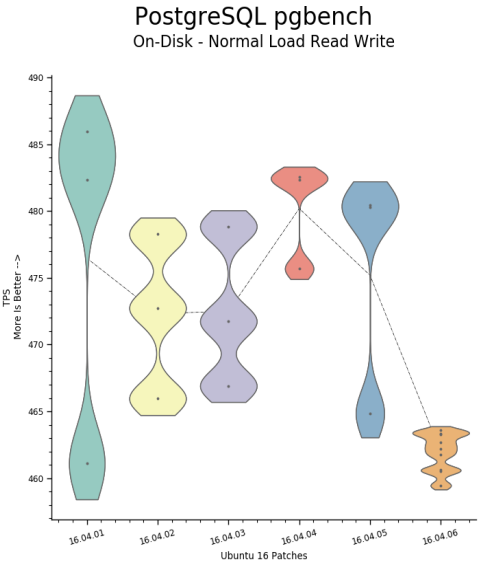



(c) Ubuntu 16.04.06 Cores

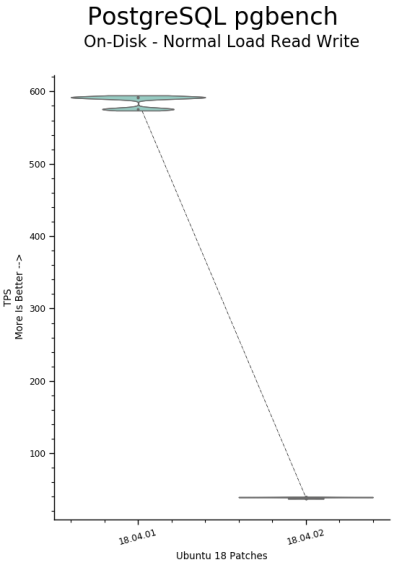



(d) Ubuntu 18.04.2 Cores

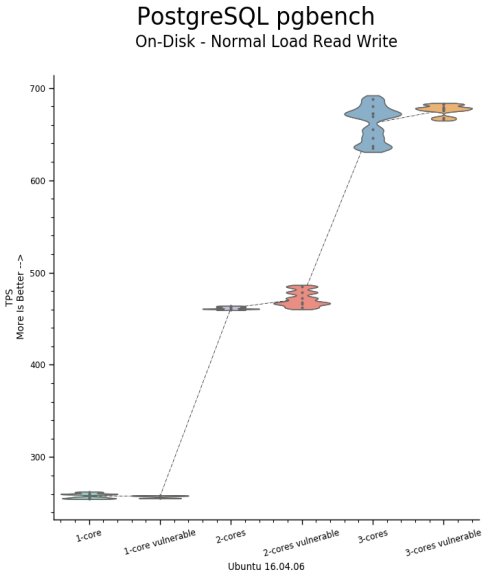
**Figure A.21: pgbench On-Disk - Normal Load - Read Only Results**



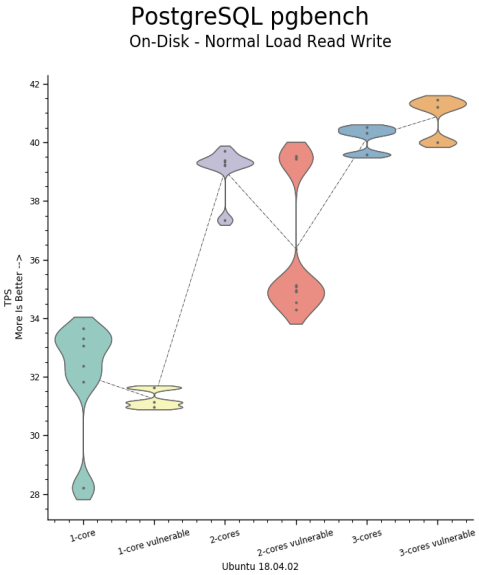

(a) 16 Patches




(b) 18 Patches

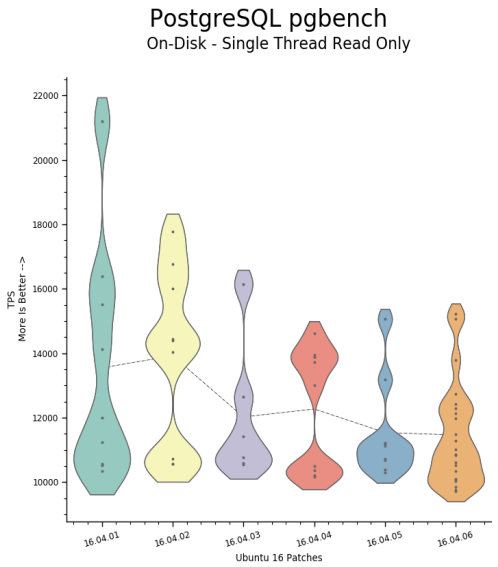



(c) Ubuntu 16.04.06 Cores

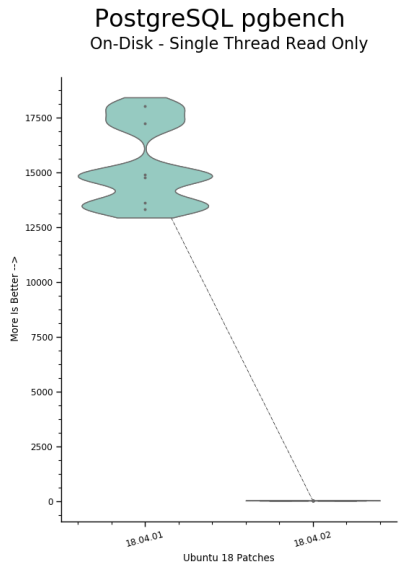



(d) Ubuntu 18.04.2 Cores

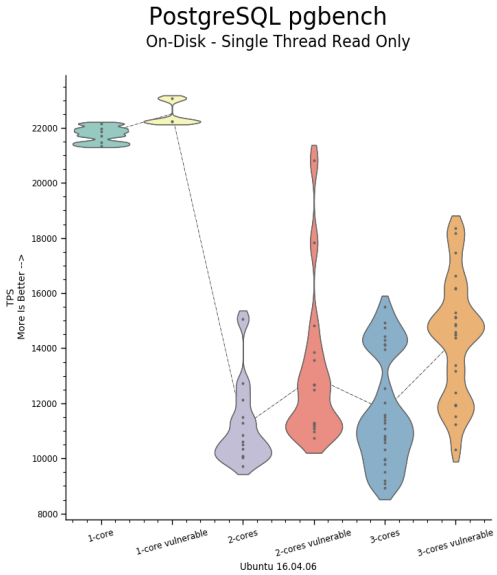
**Figure A.22: pgbench On-Disk - Normal Load - Read Write Results**



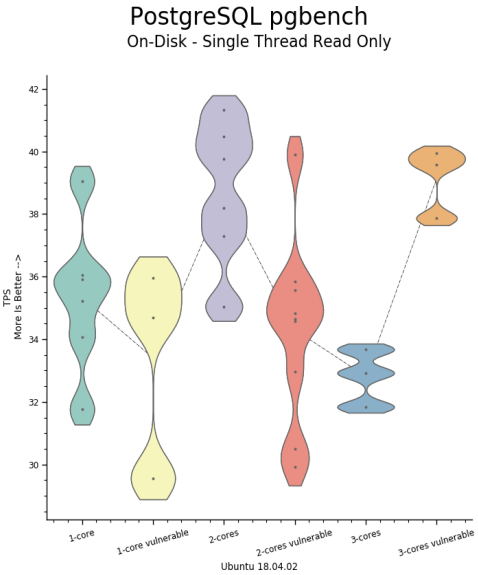

(a) 16 Patches




(b) 18 Patches

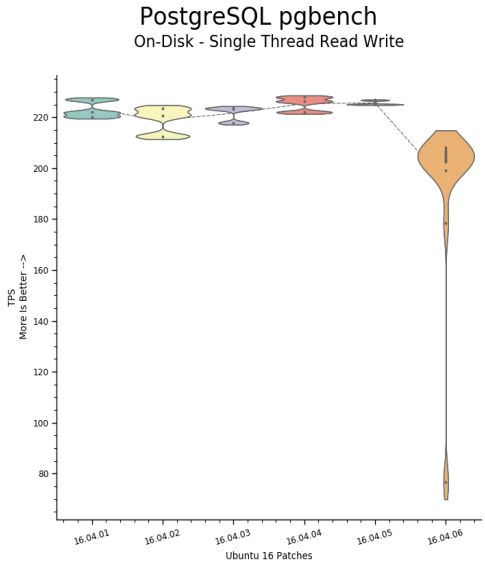



(c) Ubuntu 16.04.06 Cores

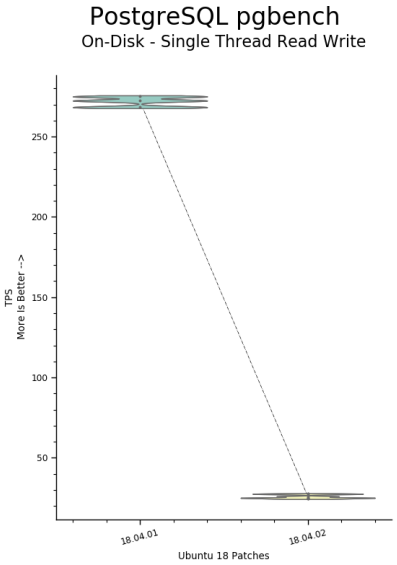



(d) Ubuntu 18.04.2 Cores

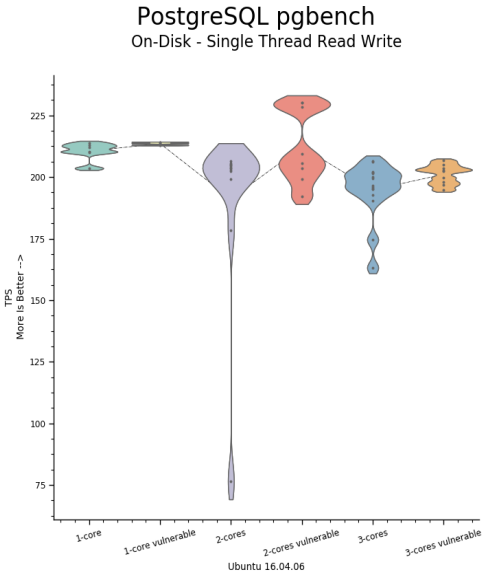
**Figure A.23: pgbench On-Disk - Single Thread - Read Only Results**



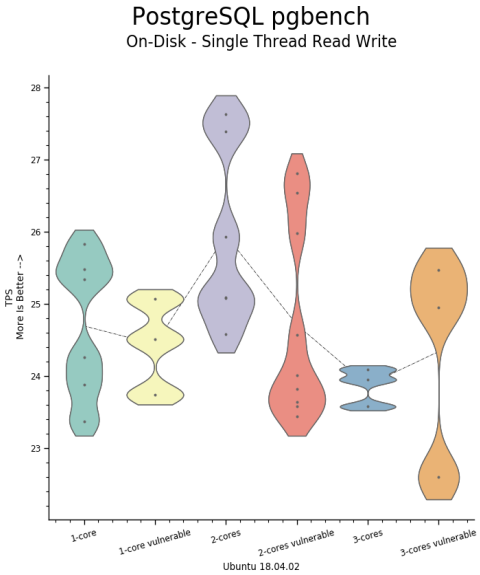

(a) 16 Patches




(b) 18 Patches

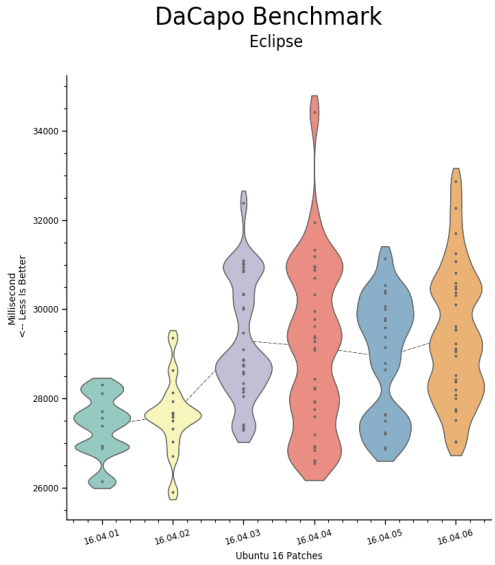



(c) Ubuntu 16.04.06 Cores

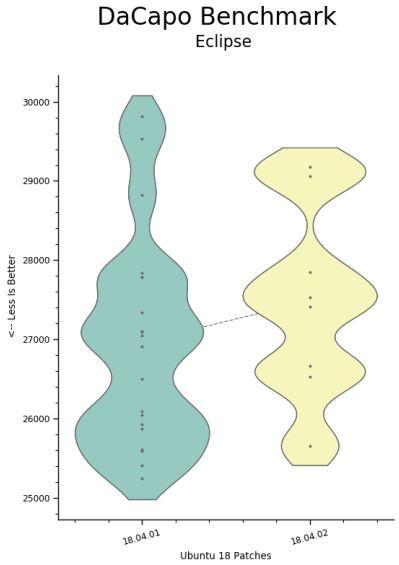



(d) Ubuntu 18.04.2 Cores

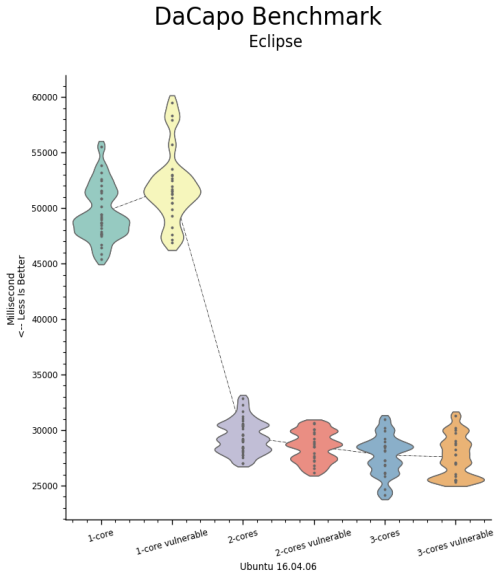
**Figure A.24: pgbench On-Disk - Single Thread - Read Write Results**



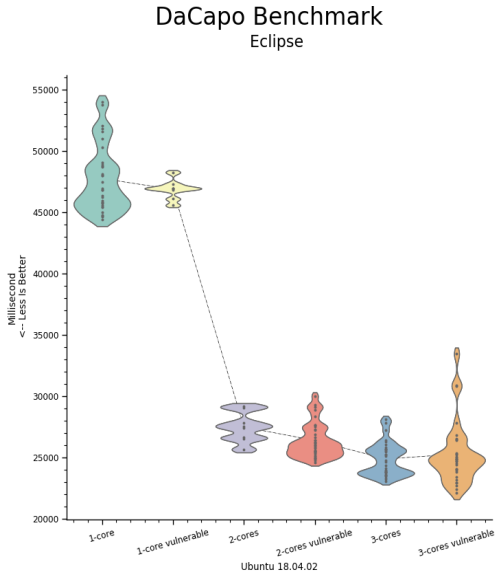

(a) 16 Patches




(b) 18 Patches

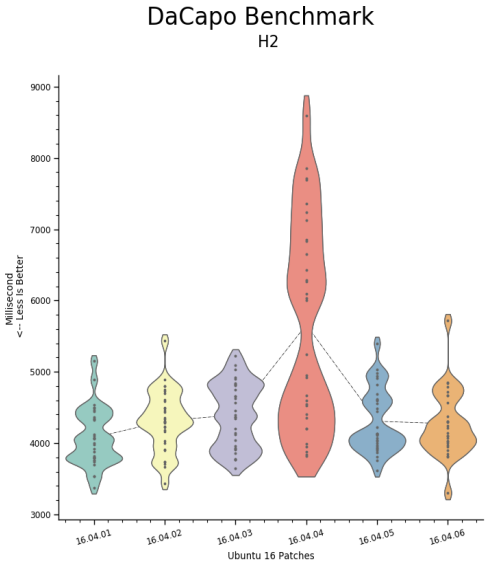



(c) Ubuntu 16.04.06 Cores

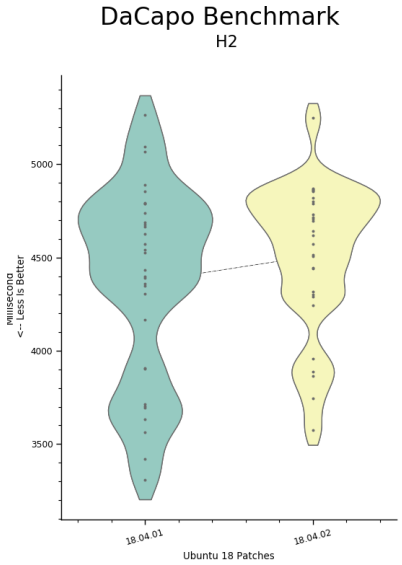



(d) Ubuntu 18.04.2 Cores

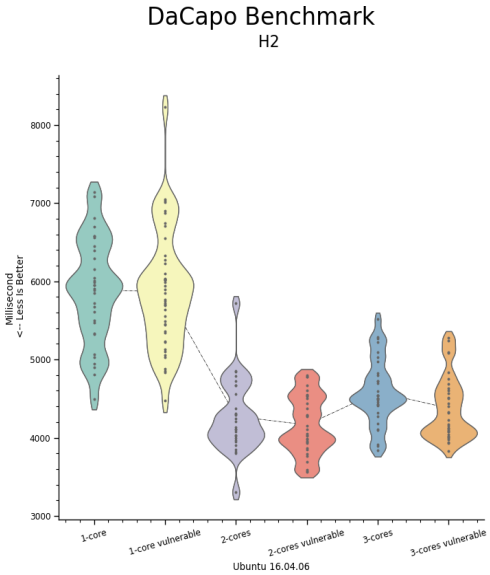
**Figure A.25: DaCapo Eclipse Results**



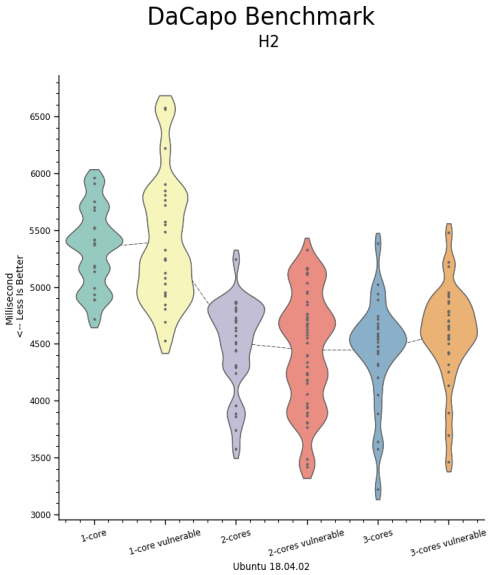

(a) 16 Patches




(b) 18 Patches

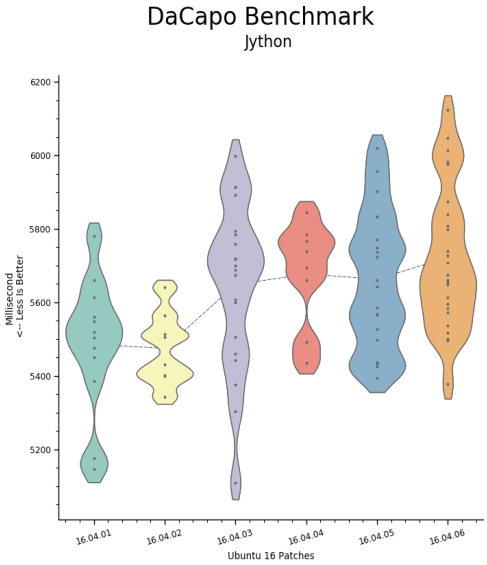



(c) Ubuntu 16.04.06 Cores

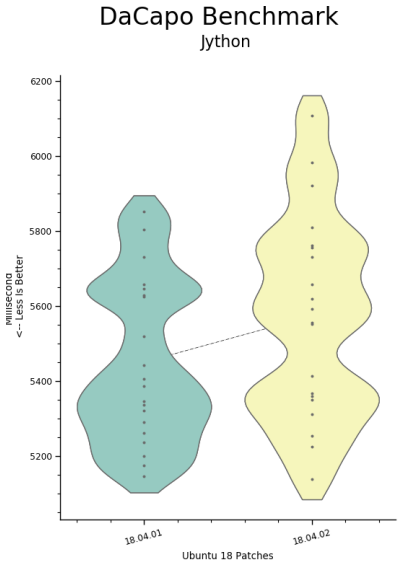



(d) Ubuntu 18.04.2 Cores

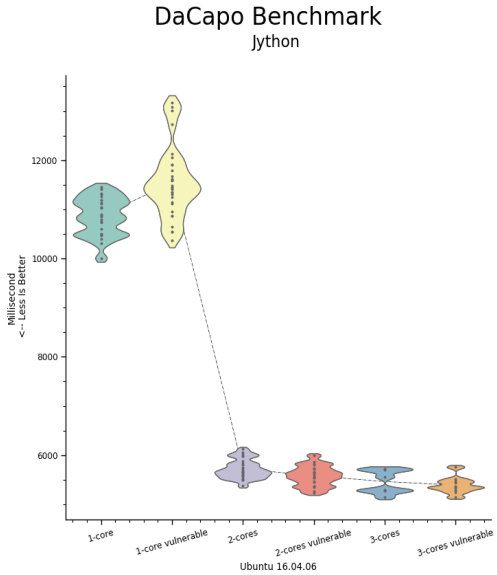
**Figure A.26: DaCapo H2 Results**



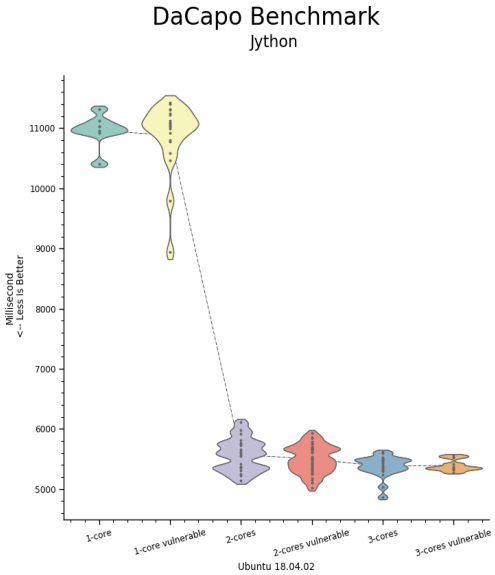

(a) 16 Patches




(b) 18 Patches

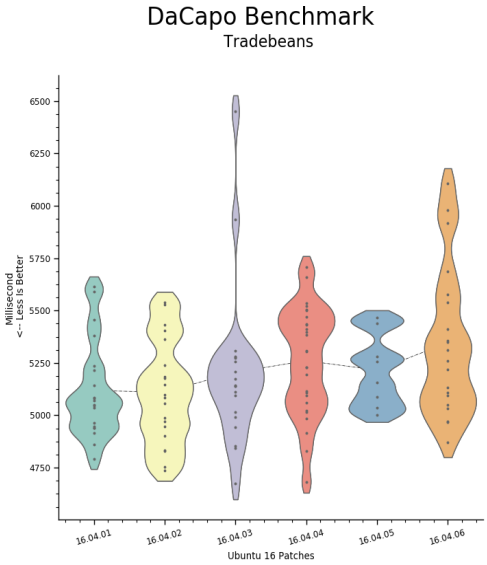



(c) Ubuntu 16.04.06 Cores

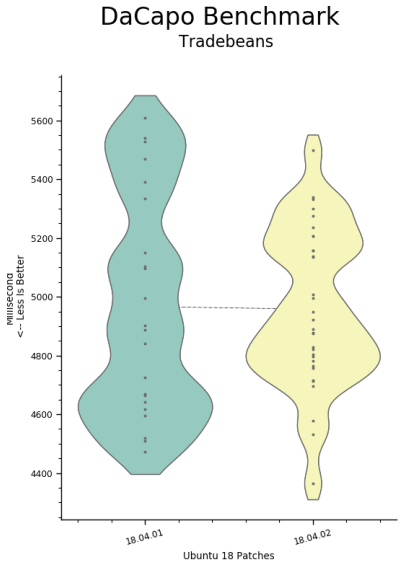



(d) Ubuntu 18.04.2 Cores

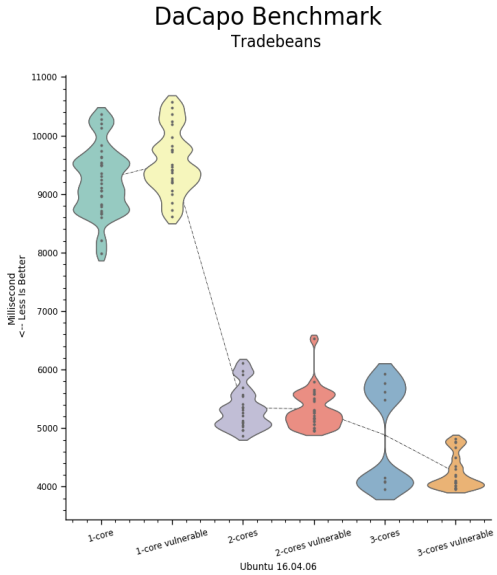
**Figure A.27: DaCapo Jython Results**



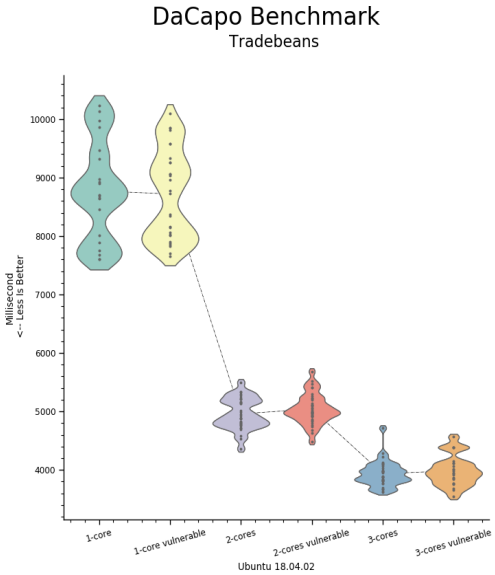

(a) 16 Patches




(b) 18 Patches

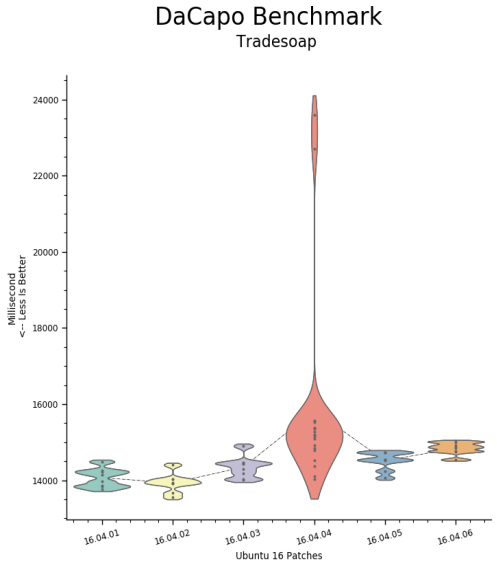



(c) Ubuntu 16.04.06 Cores

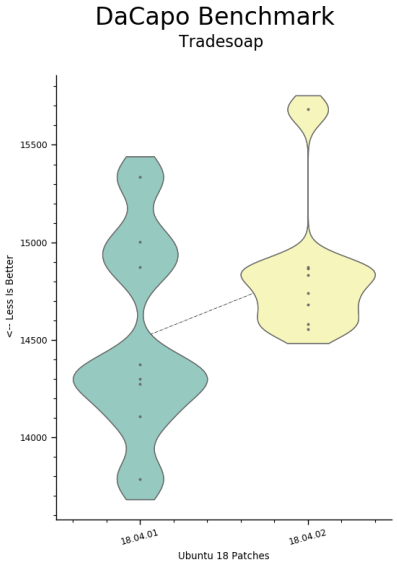



(d) Ubuntu 18.04.2 Cores

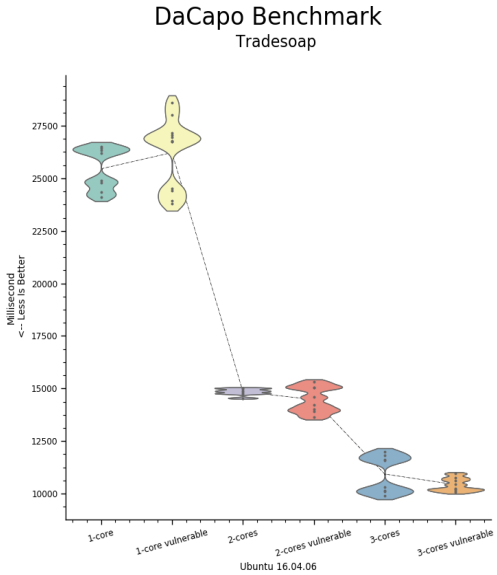
**Figure A.28: DaCapo Tradebeans Results**



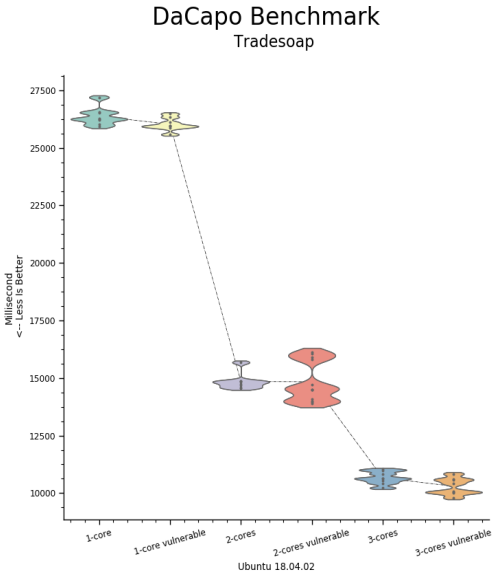

(a) 16 Patches




(b) 18 Patches

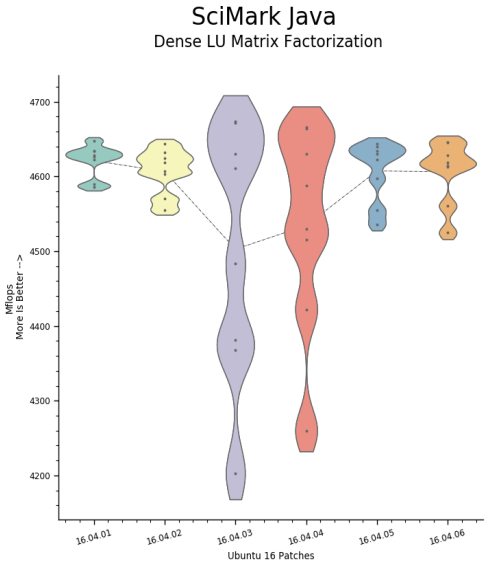



(c) Ubuntu 16.04.06 Cores

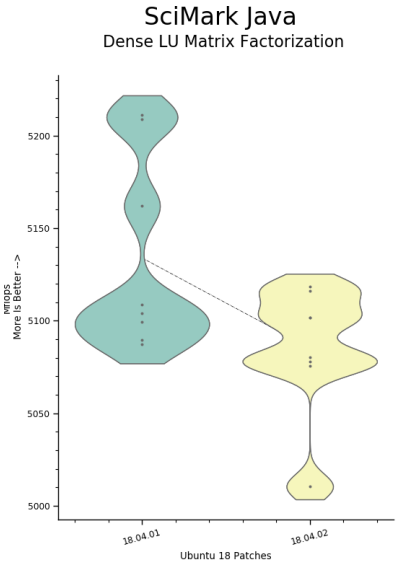



(d) Ubuntu 18.04.2 Cores

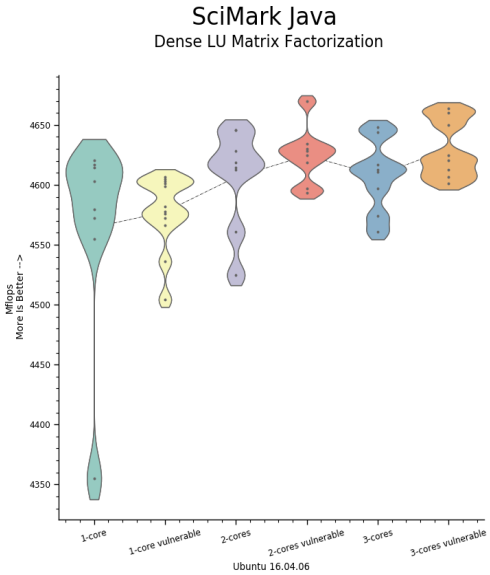
**Figure A.29: DaCapo Tradesoap Results**



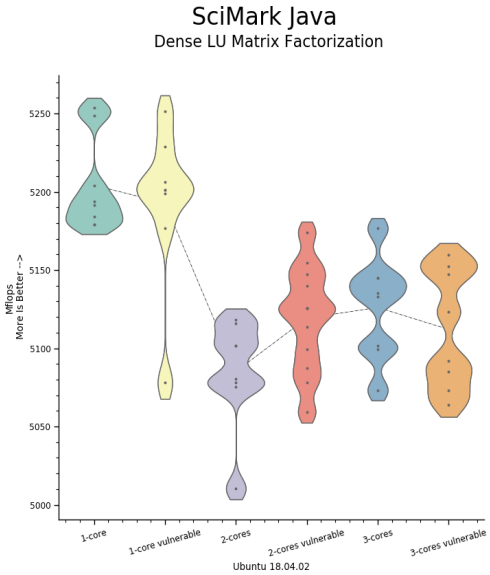

(a) 16 Patches




(b) 18 Patches

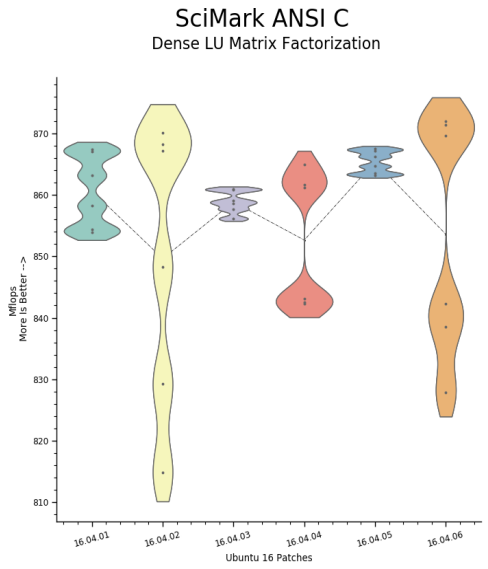



(c) Ubuntu 16.04.06 Cores

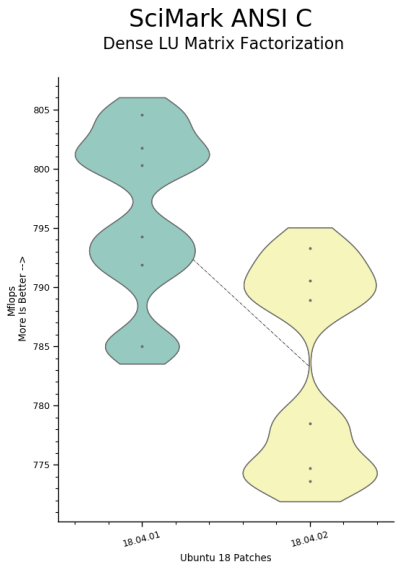



(d) Ubuntu 18.04.2 Cores

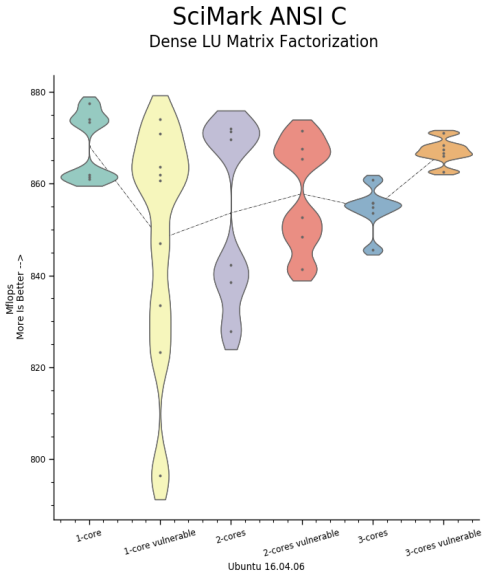
**Figure A.30: SciMark: Java - Dense LU Matrix Factorization Results**



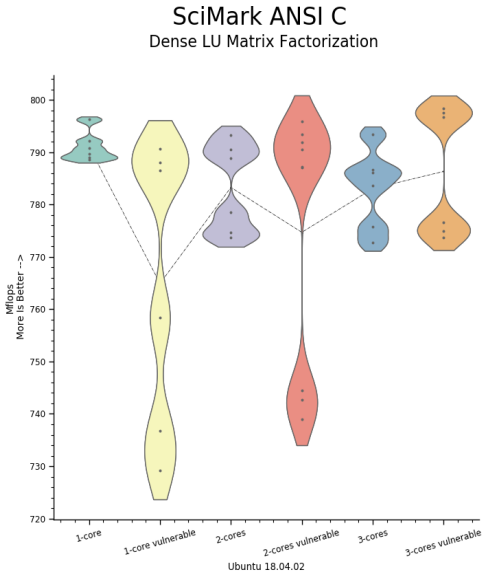

(a) 16 Patches




(b) 18 Patches

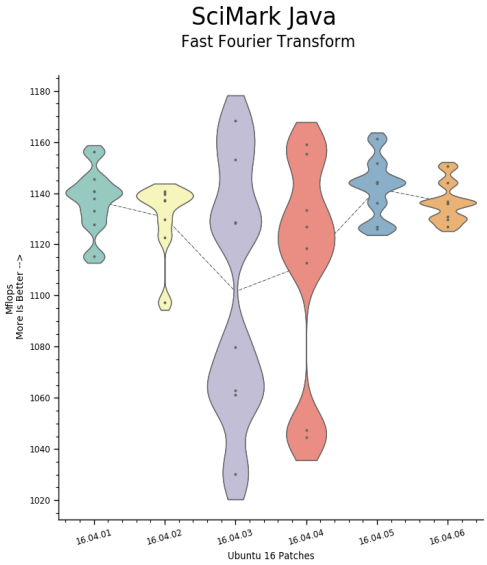



(c) Ubuntu 16.04.06 Cores

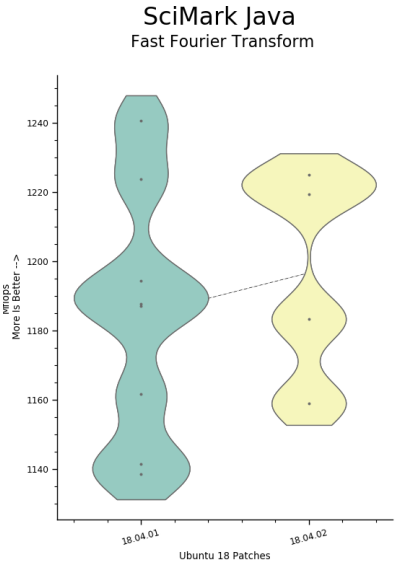



(d) Ubuntu 18.04.2 Cores

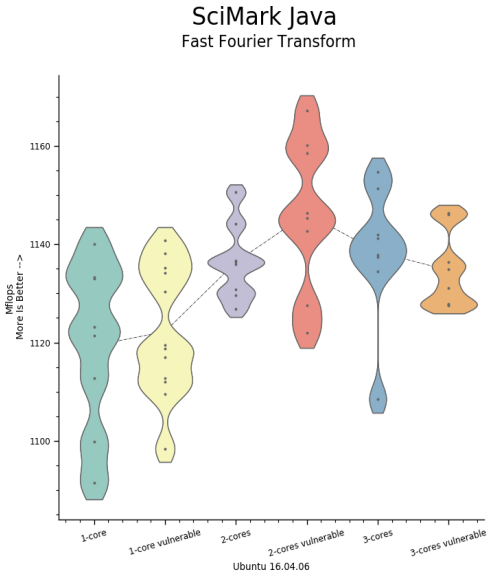
**Figure A.31: SciMark: ANSI C - Dense LU Matrix Factorization Results**



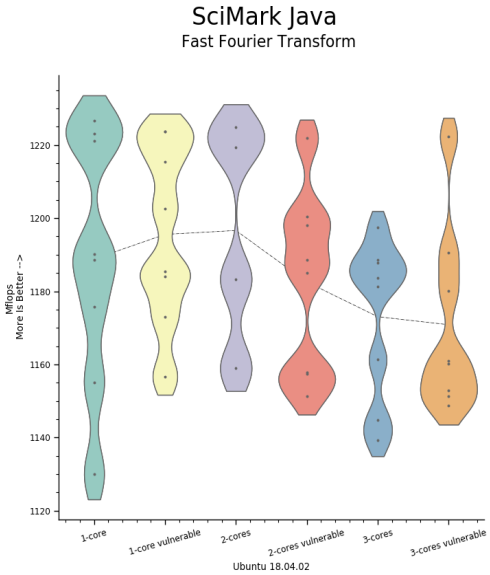

(a) 16 Patches




(b) 18 Patches

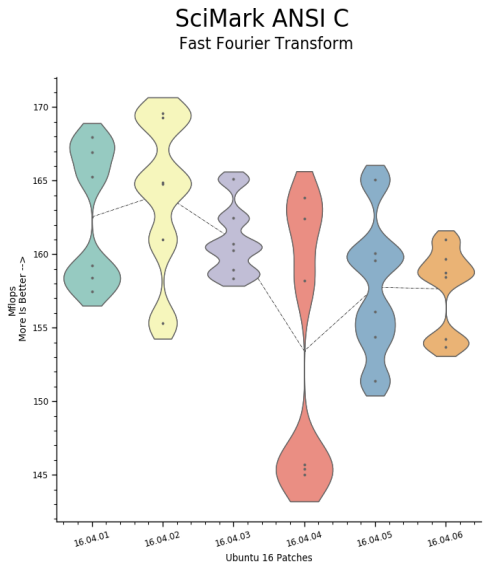



(c) Ubuntu 16.04.06 Cores

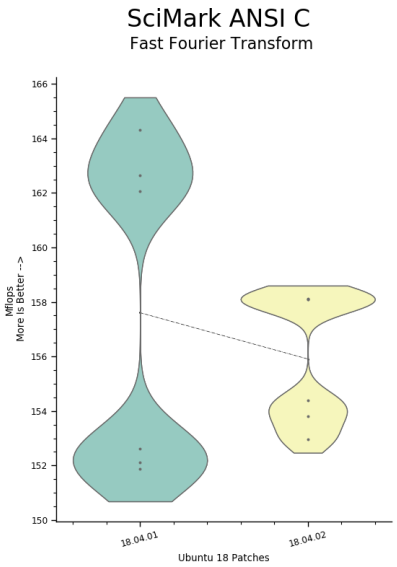



(d) Ubuntu 18.04.2 Cores

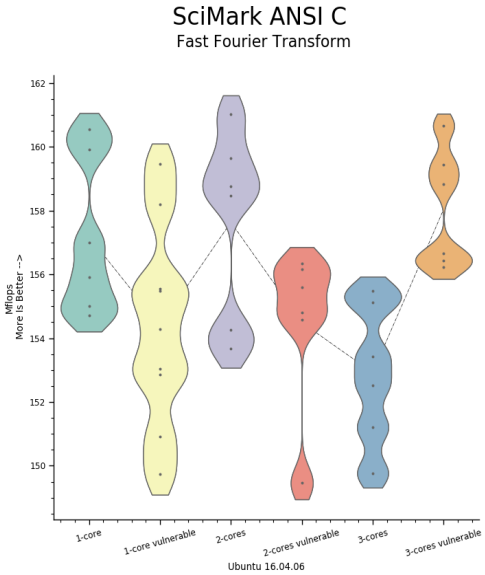
**Figure A.32: SciMark: Java - Fast Fourier Transform Results**



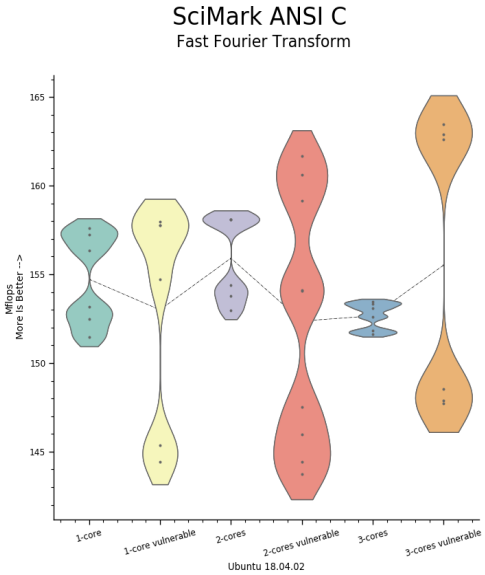

(a) 16 Patches




(b) 18 Patches

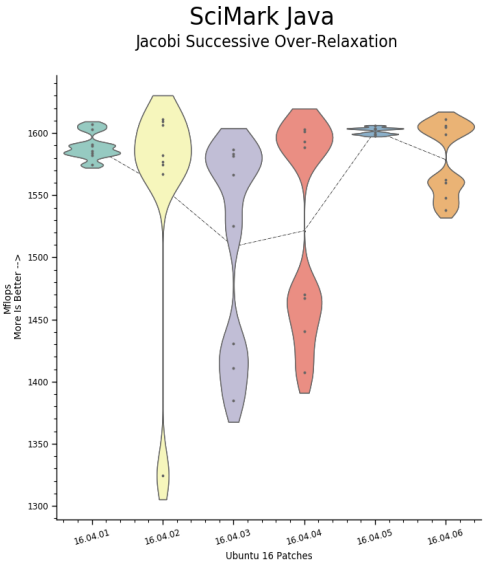



(c) Ubuntu 16.04.06 Cores

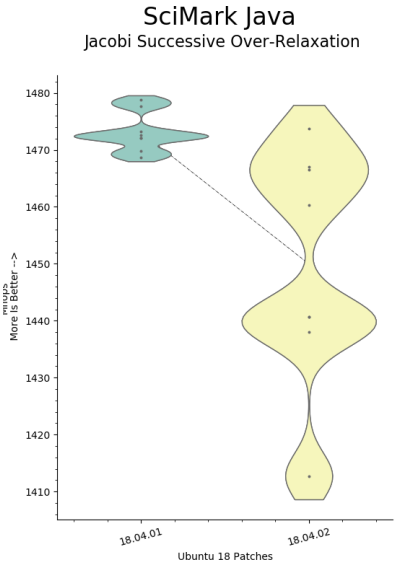



(d) Ubuntu 18.04.2 Cores

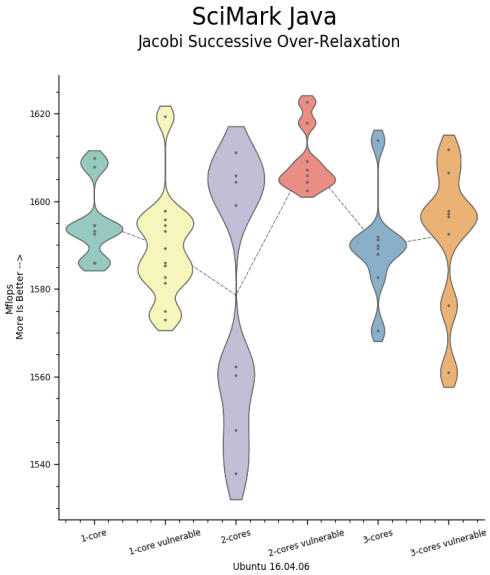
**Figure A.33: SciMark: ANSI C - Fast Fourier Transform Results**



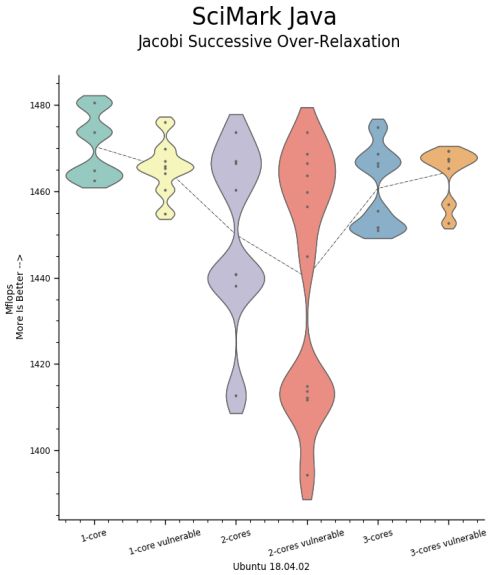

(a) 16 Patches




(b) 18 Patches

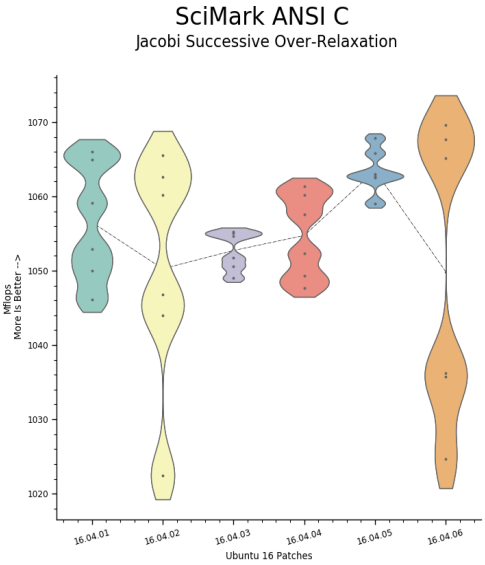



(c) Ubuntu 16.04.06 Cores

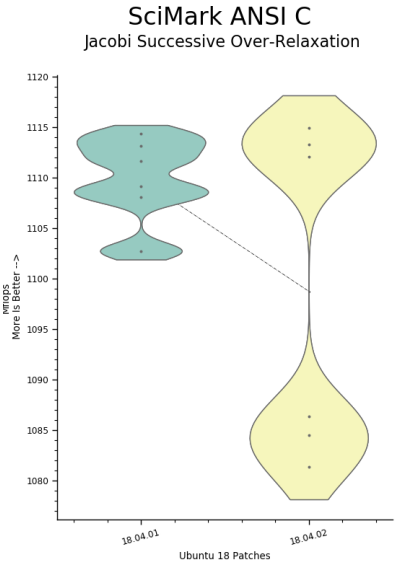



(d) Ubuntu 18.04.2 Cores

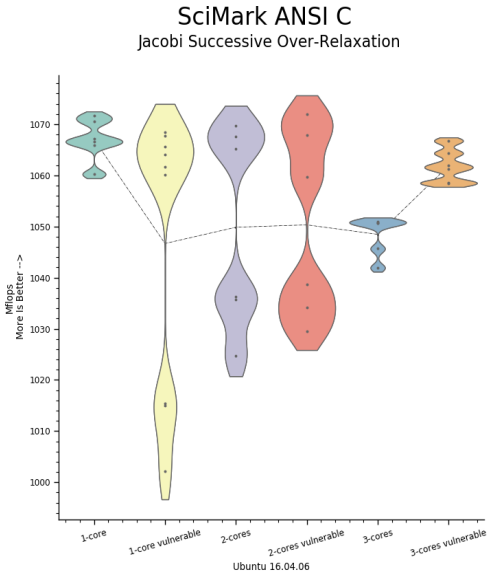
**Figure A.34: SciMark: Java - Jacobi Successive Over-Relaxation Results**



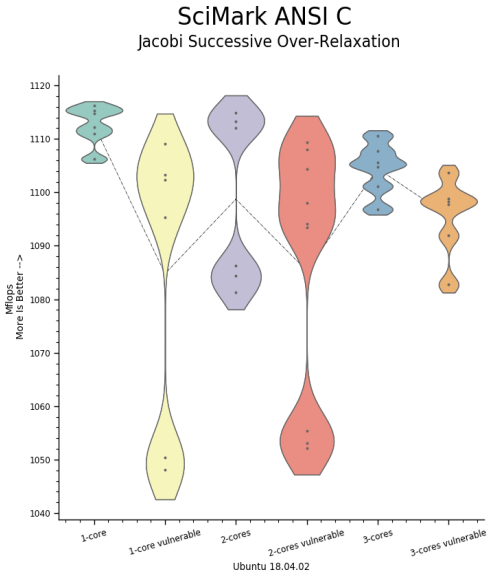

(a) 16 Patches




(b) 18 Patches

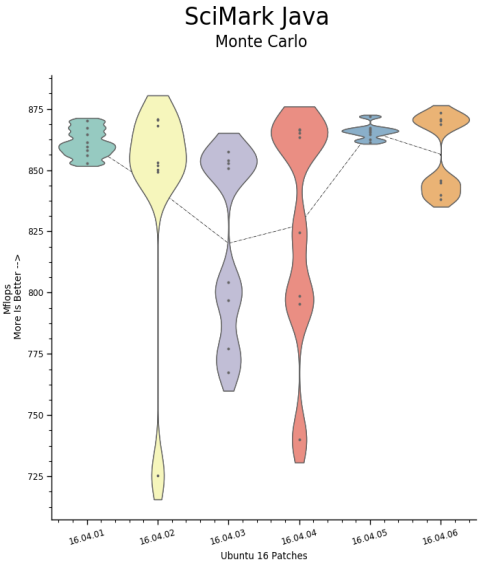



(c) Ubuntu 16.04.06 Cores

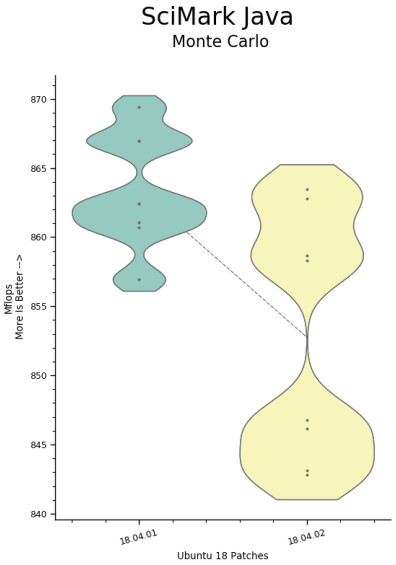



(d) Ubuntu 18.04.2 Cores

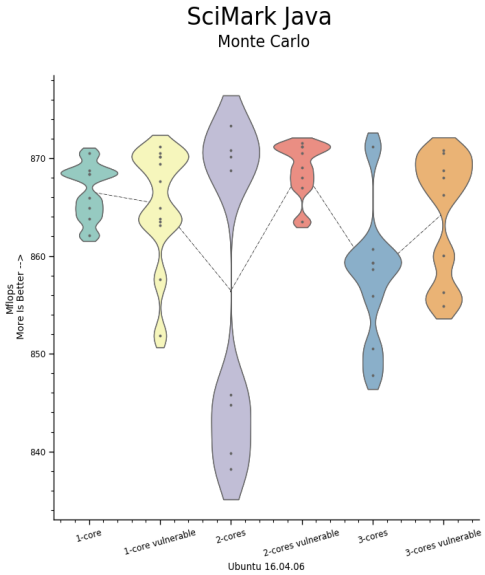
**Figure A.35: SciMark: ANSI C - Jacobi Successive Over-Relaxation Results**



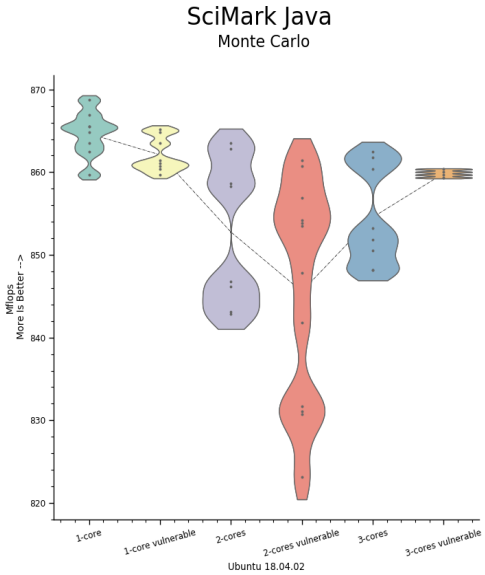

(a) 16 Patches




(b) 18 Patches

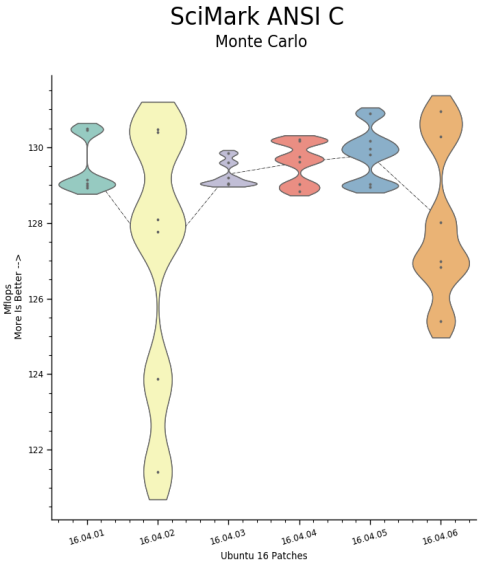



(c) Ubuntu 16.04.06 Cores

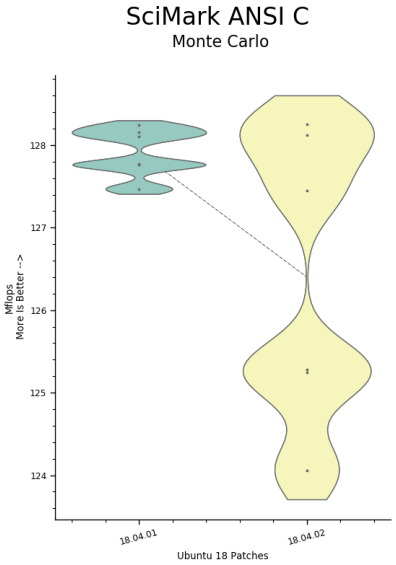



(d) Ubuntu 18.04.2 Cores

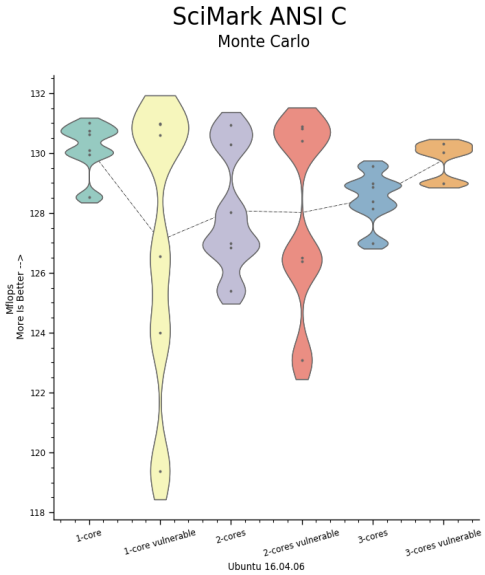
**Figure A.36: SciMark: Java - Monte Carlo Results**



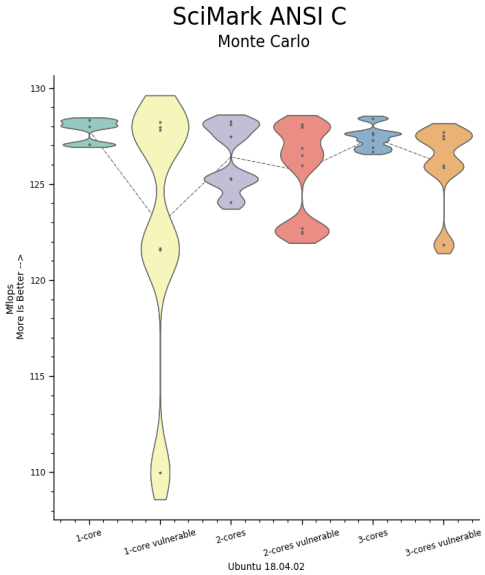

(a) 16 Patches




(b) 18 Patches

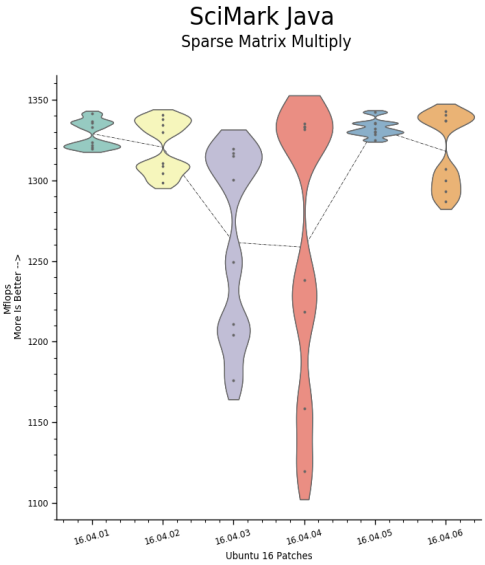



(c) Ubuntu 16.04.06 Cores

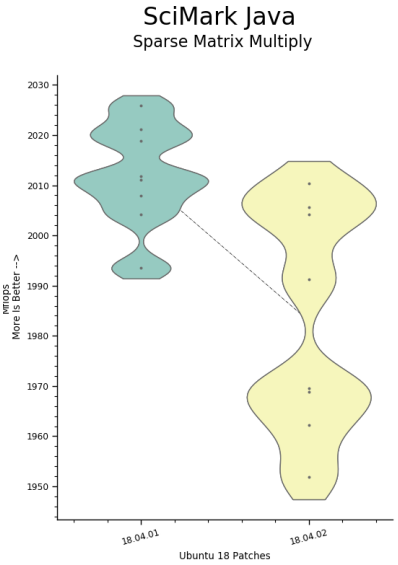



(d) Ubuntu 18.04.2 Cores

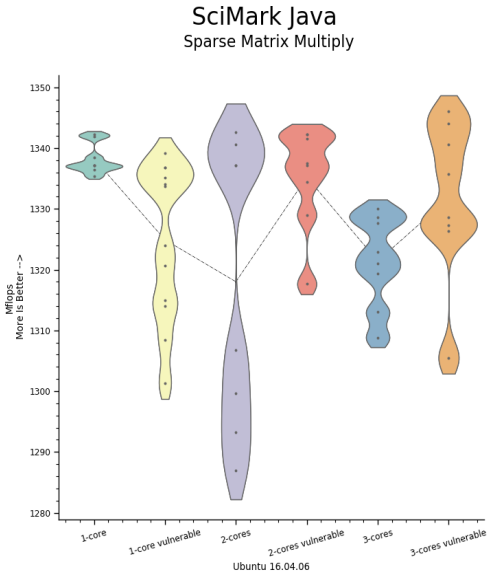
**Figure A.37: SciMark: ANSI C - Monte Carlo Results**



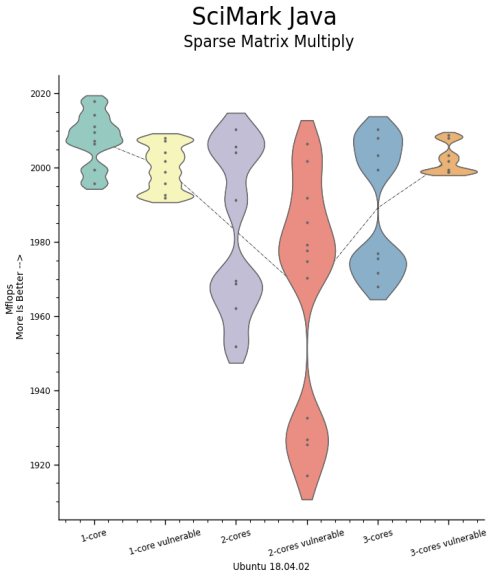

(a) 16 Patches




(b) 18 Patches

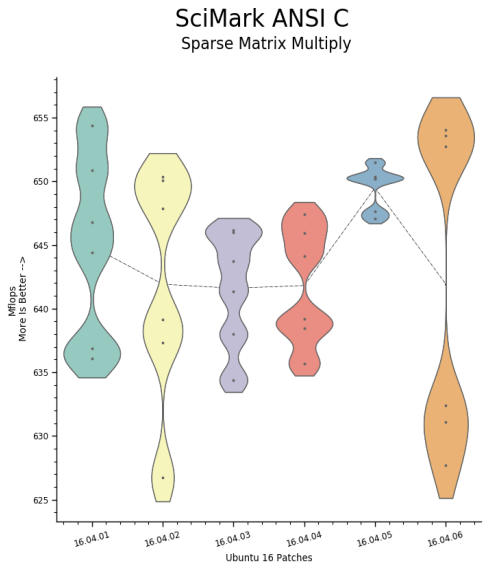



(c) Ubuntu 16.04.06 Cores

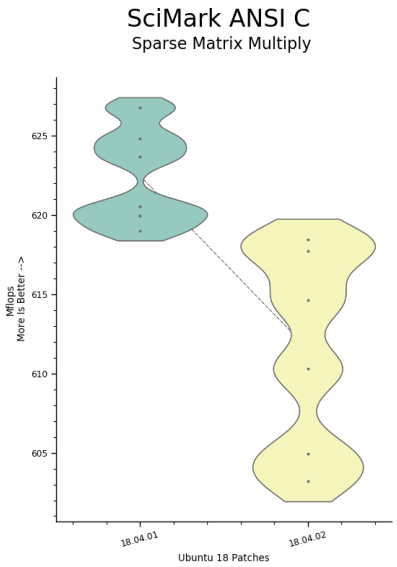



(d) Ubuntu 18.04.2 Cores

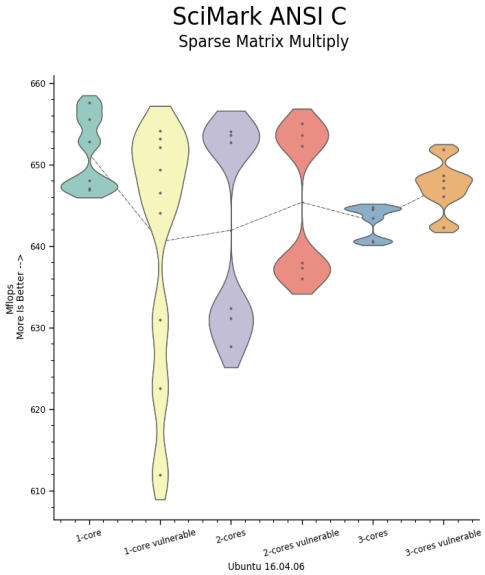
**Figure A.38: SciMark: Java - Sparse Matrix Multiply Results**



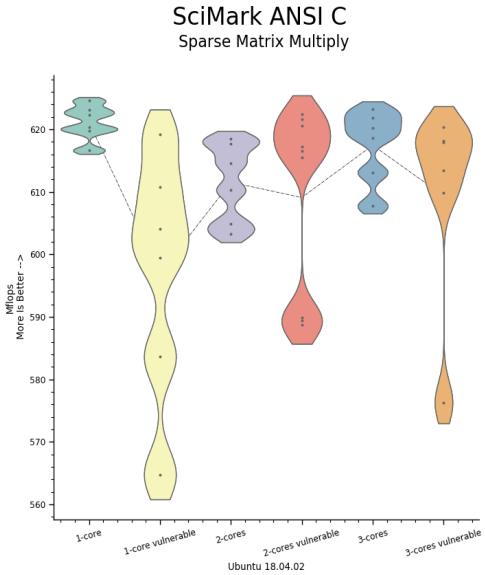

(a) 16 Patches




(b) 18 Patches

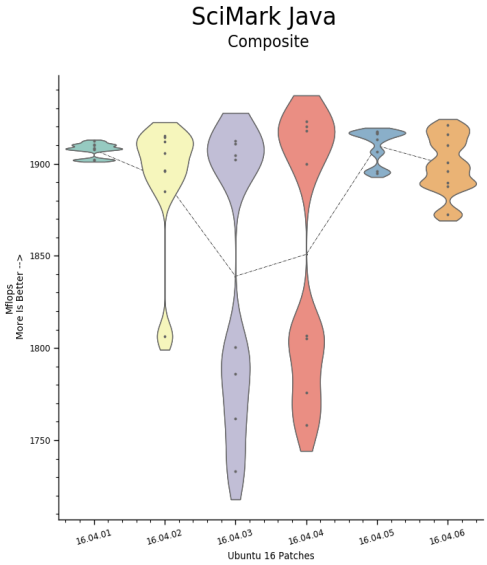



(c) Ubuntu 16.04.06 Cores

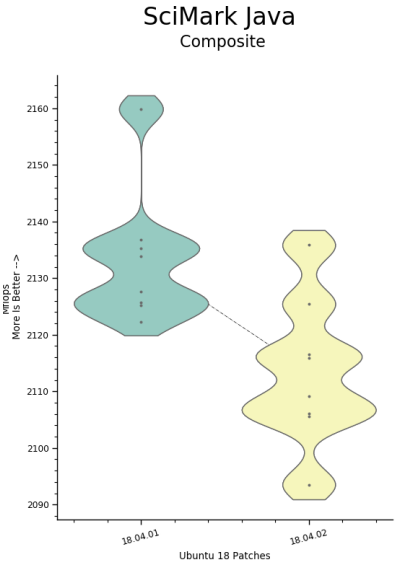



(d) Ubuntu 18.04.2 Cores

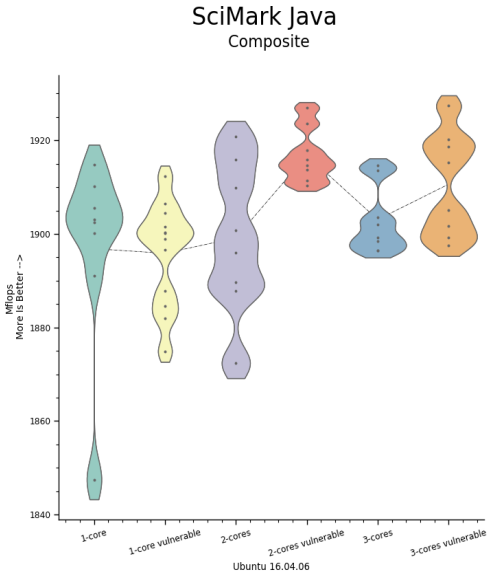
**Figure A.39: SciMark: ANSI C - Sparse Matrix Multiply Results**



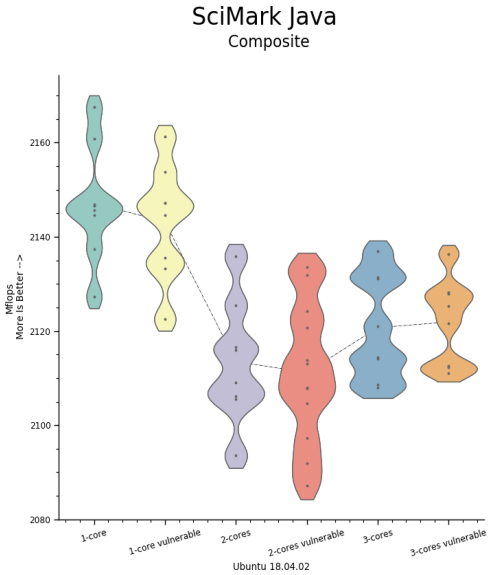

(a) 16 Patches




(b) 18 Patches

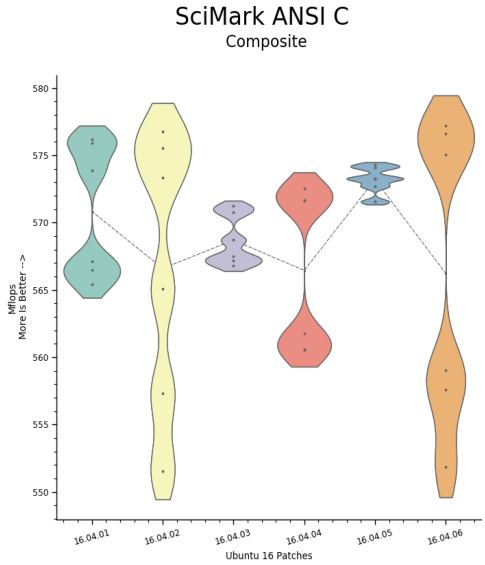



(c) Ubuntu 16.04.06 Cores

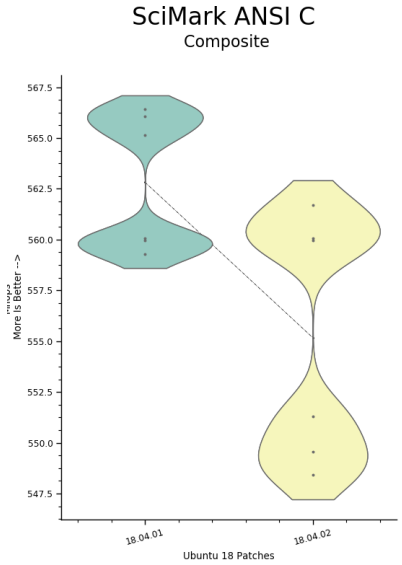



(d) Ubuntu 18.04.2 Cores

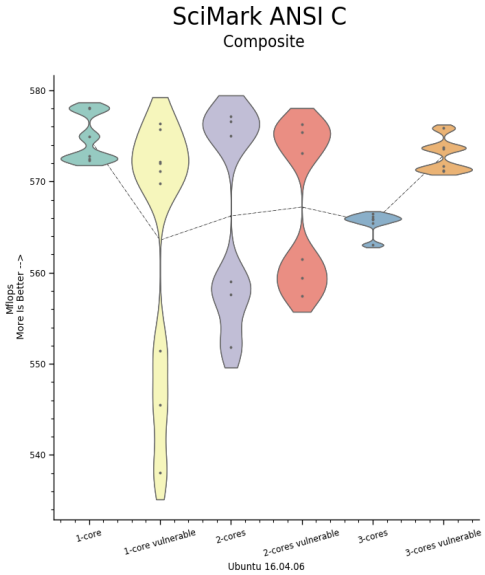
**Figure A.40: SciMark: Java - Composite Results**



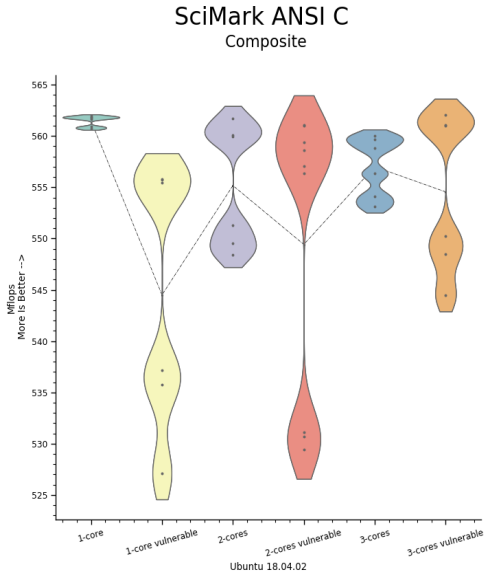

(a) 16 Patches




(b) 18 Patches

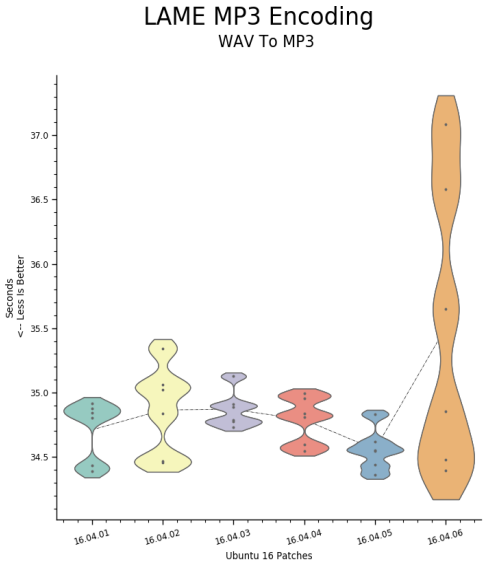



(c) Ubuntu 16.04.06 Cores

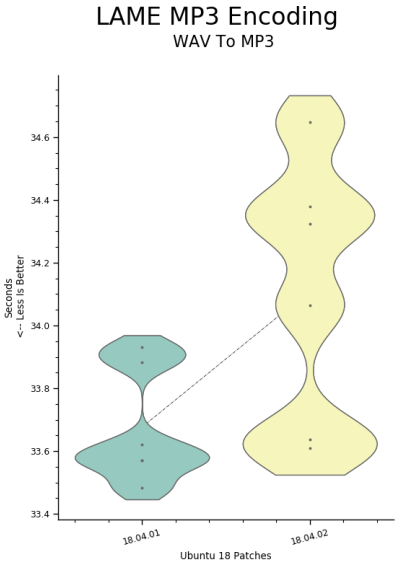



(d) Ubuntu 18.04.2 Cores

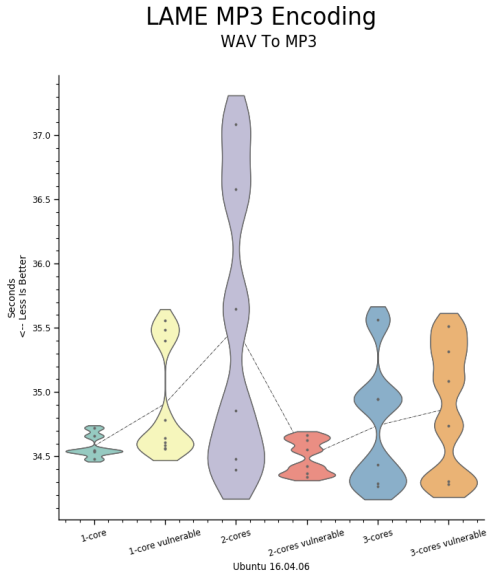
**Figure A.41: SciMark: ANSI C - Composite Results**



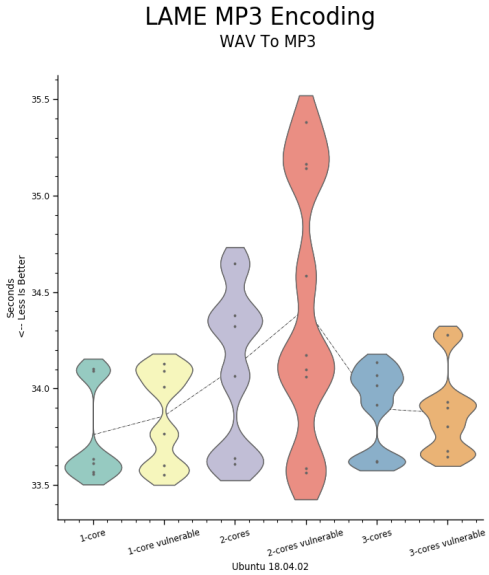

(a) 16 Patches




(b) 18 Patches

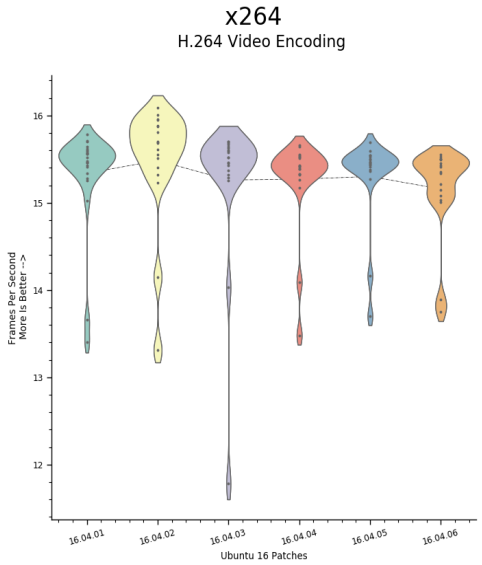



(c) Ubuntu 16.04.06 Cores

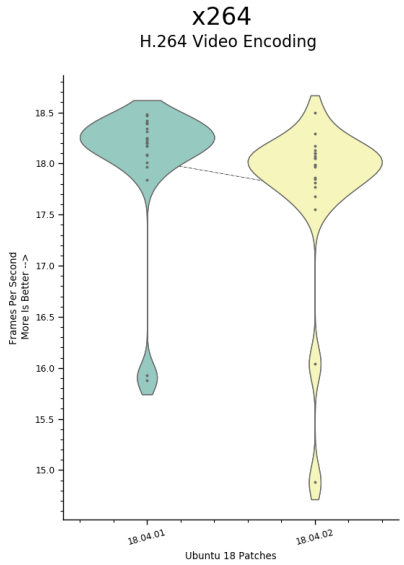



(d) Ubuntu 18.04.2 Cores

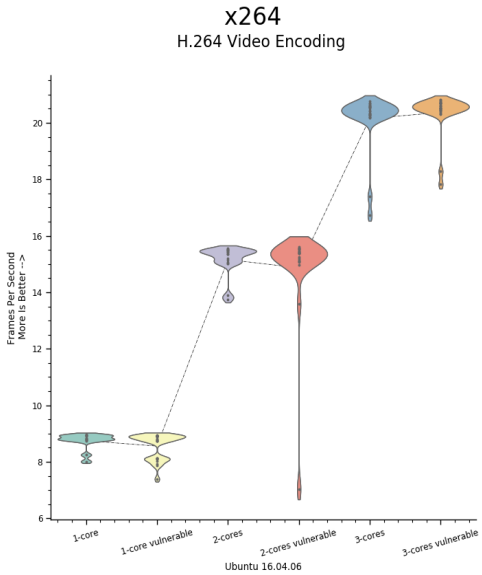
**Figure A.42: LAME MP3 Encoding: WAV To MP3 Results**



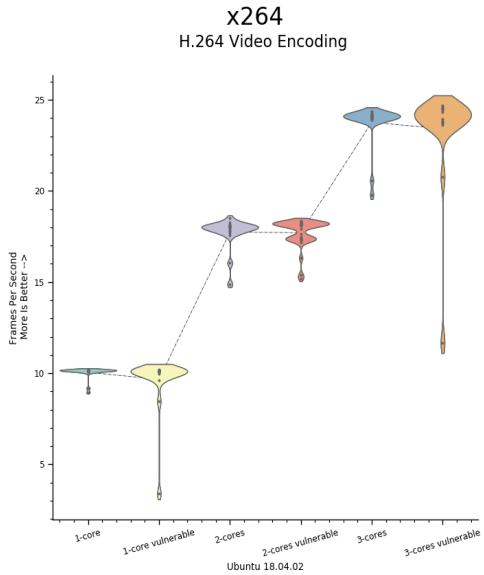

(a) 16 Patches




(b) 18 Patches

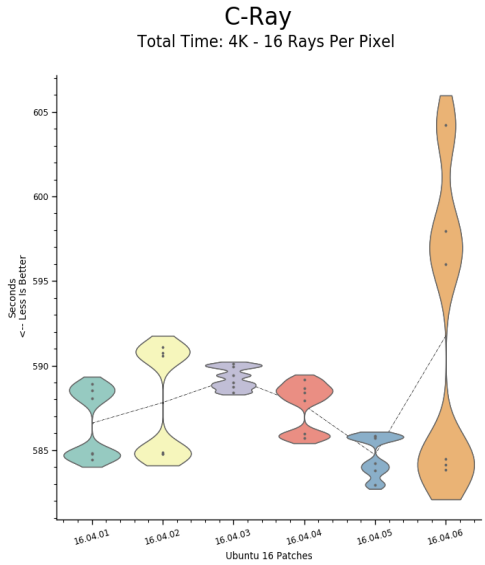



(c) Ubuntu 16.04.06 Cores

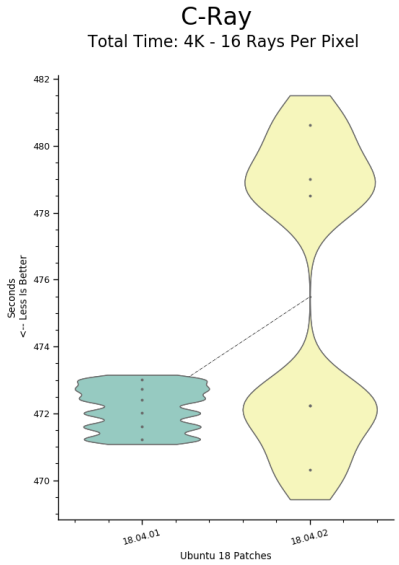



(d) Ubuntu 18.04.2 Cores

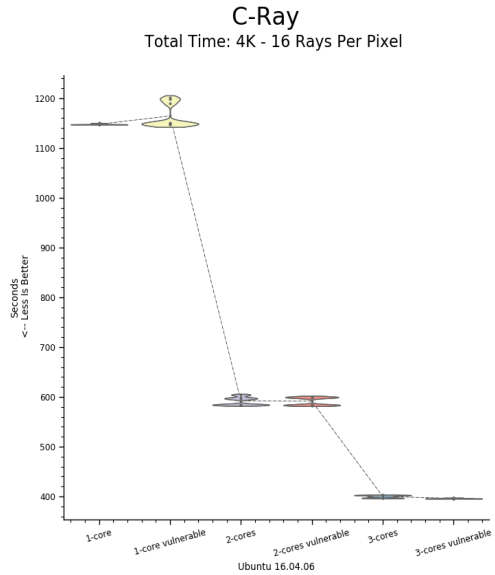
**Figure A.43: x264 Video Encoding Results**



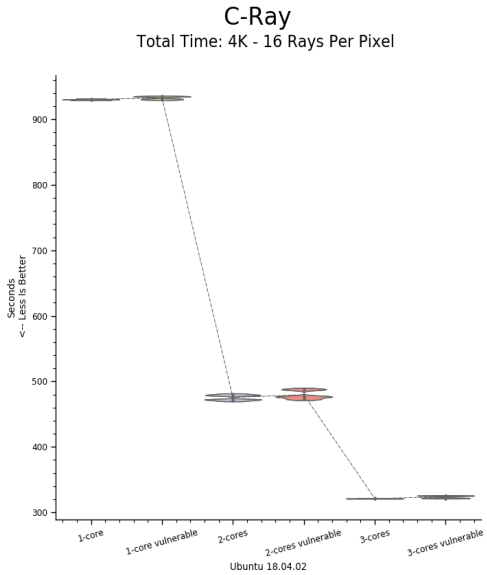

(a) 16 Patches




(b) 18 Patches

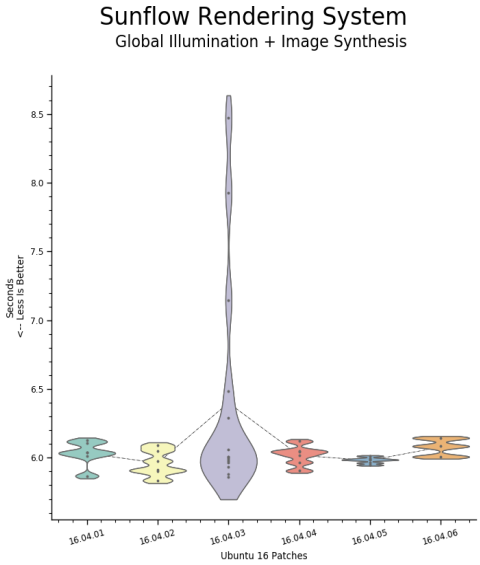



(c) Ubuntu 16.04.06 Cores

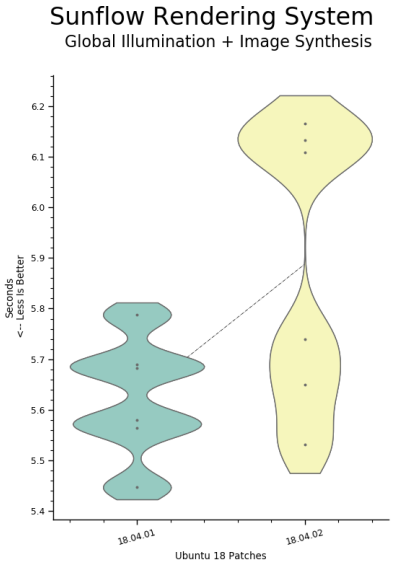



(d) Ubuntu 18.04.2 Cores

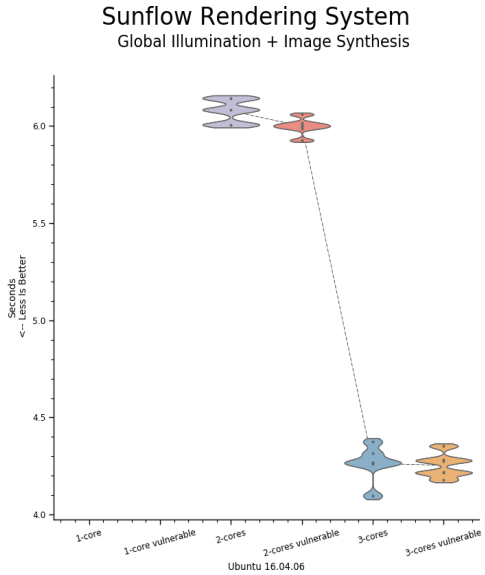
**Figure A.44: C-Ray Results**



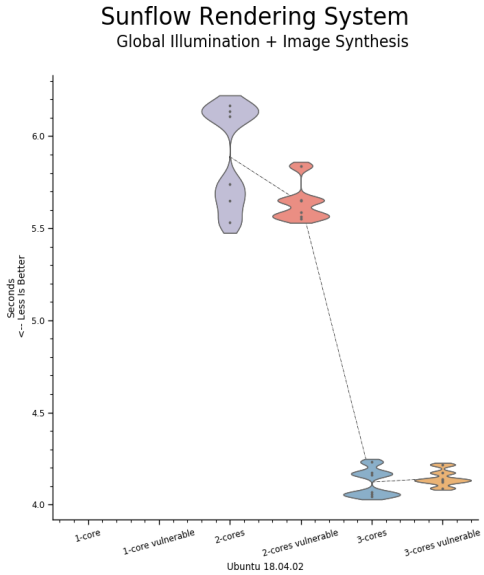

(a) 16 Patches




(b) 18 Patches

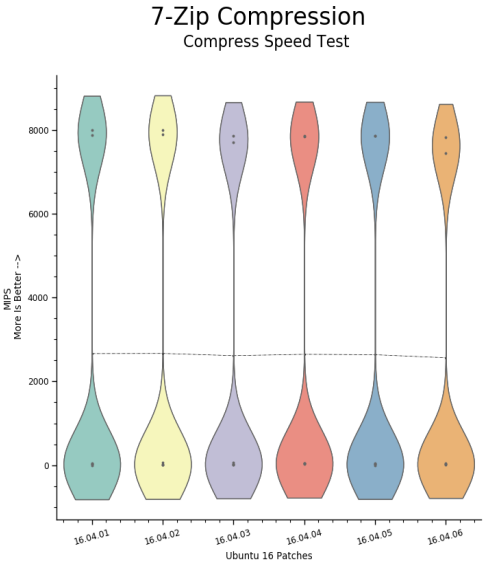



(c) Ubuntu 16.04.06 Cores

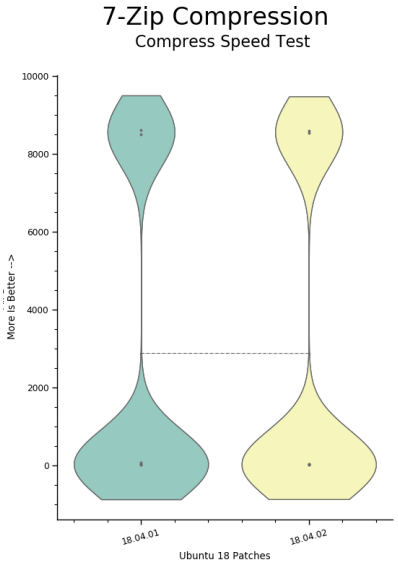



(d) Ubuntu 18.04.2 Cores

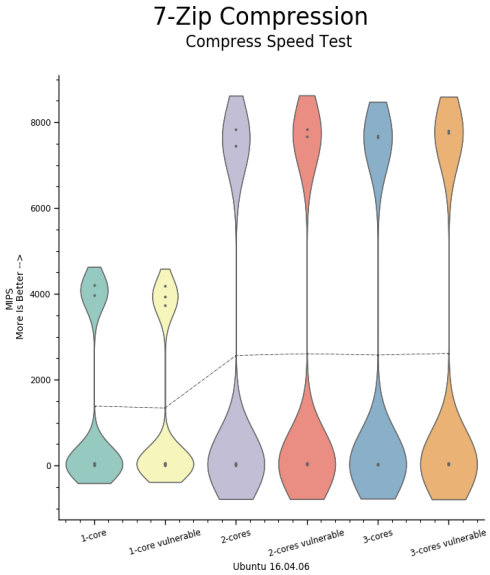
**Figure A.45: Sunflow Rendering System Results**



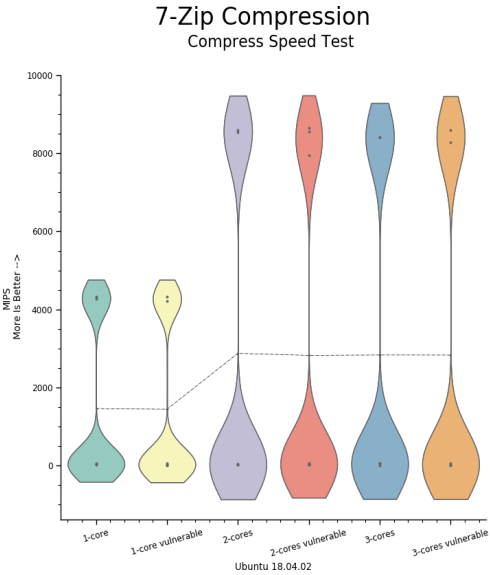

(a) 16 Patches




(b) 18 Patches

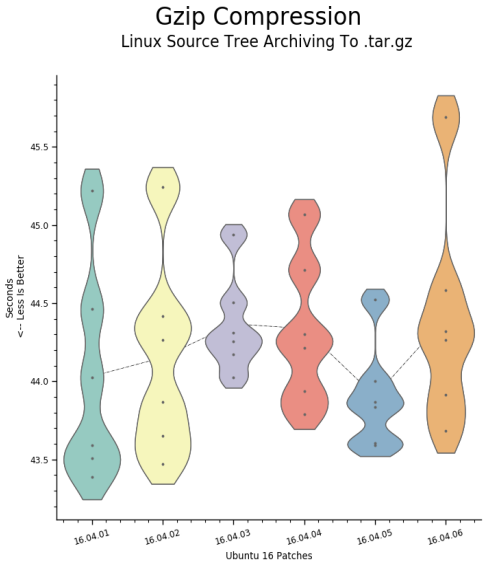



(c) Ubuntu 16.04.06 Cores

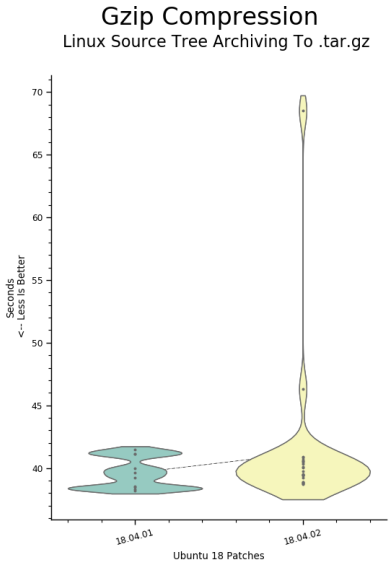



(d) Ubuntu 18.04.2 Cores

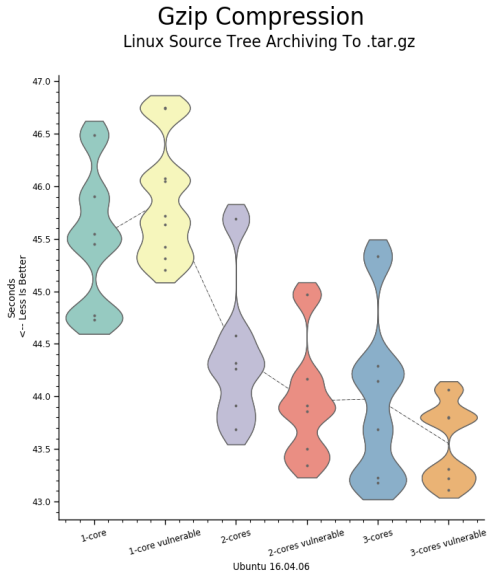
**Figure A.46: 7-zip Compression Results**



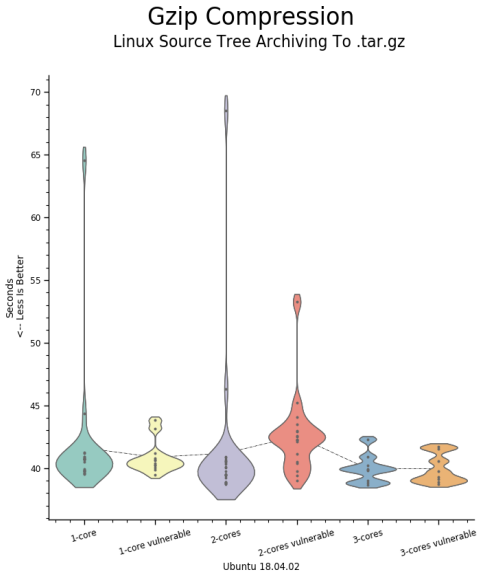

(a) 16 Patches




(b) 18 Patches

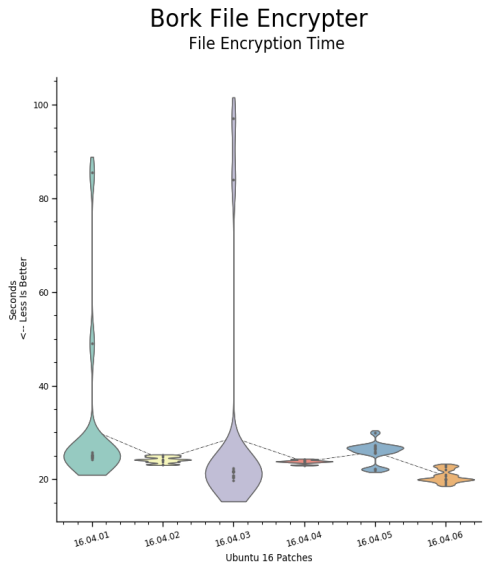



(c) Ubuntu 16.04.06 Cores

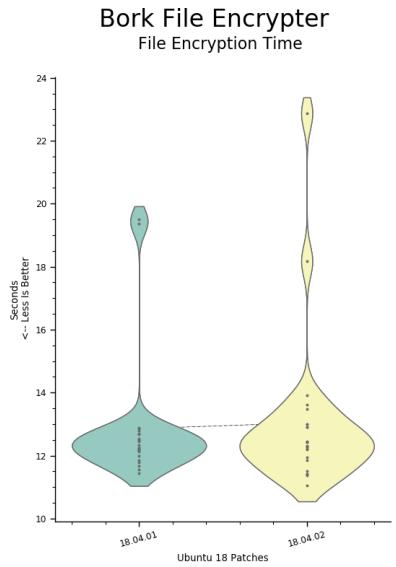



(d) Ubuntu 18.04.2 Cores

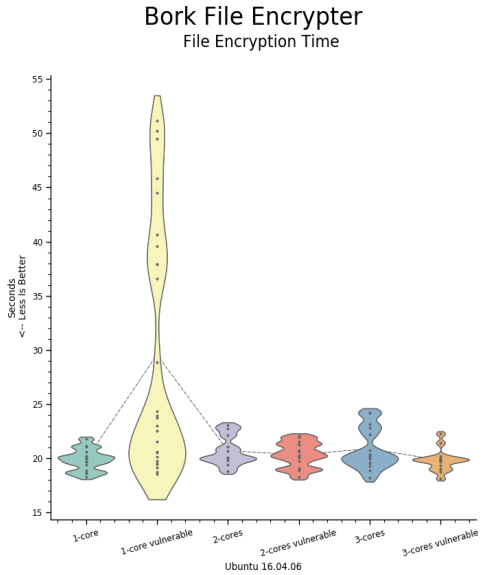
**Figure A.47: Gzip Compression Results**



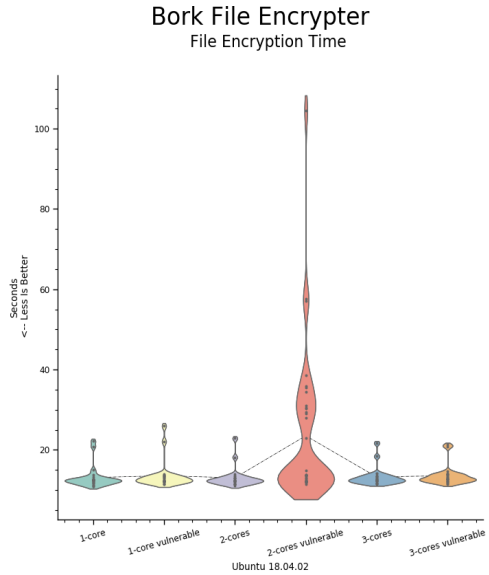

(a) 16 Patches




(b) 18 Patches

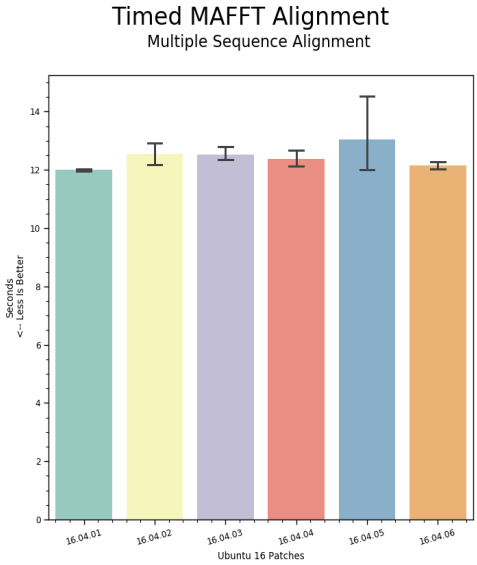



(c) Ubuntu 16.04.06 Cores

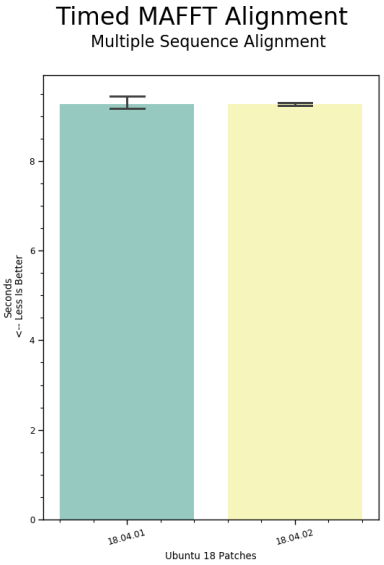



(d) Ubuntu 18.04.2 Cores

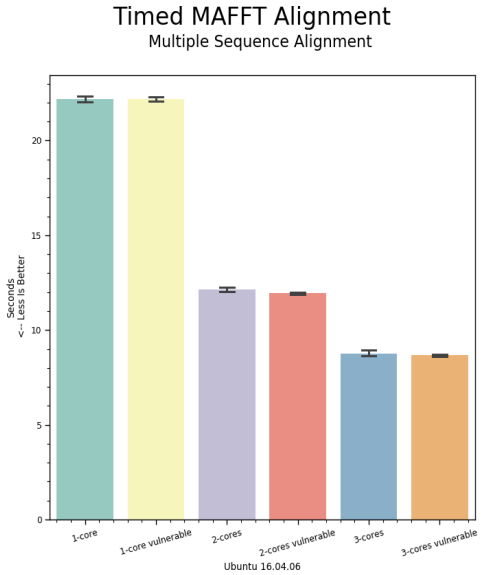
**Figure A.48: Bork Results**



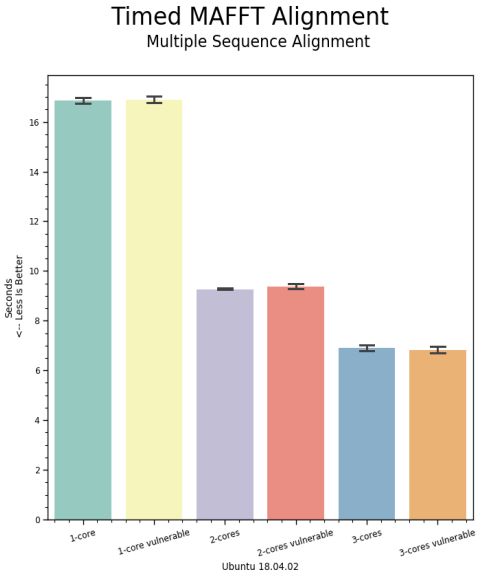

(a) 16 Patches




(b) 18 Patches

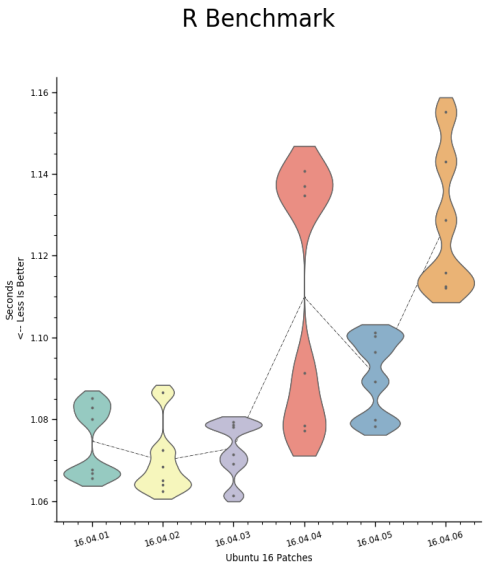



(c) Ubuntu 16.04.06 Cores

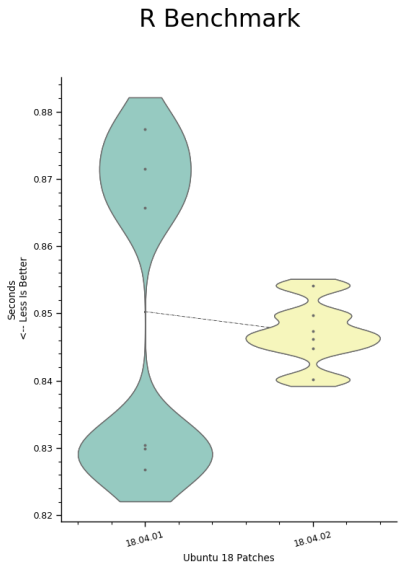



(d) Ubuntu 18.04.2 Cores

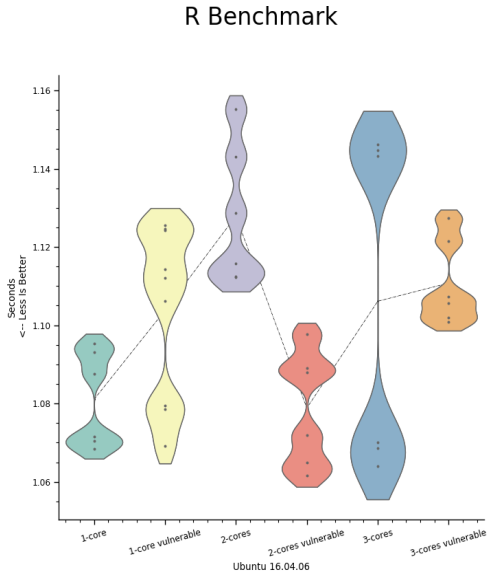
**Figure A.49: Timed MAFFT Alignment Results**



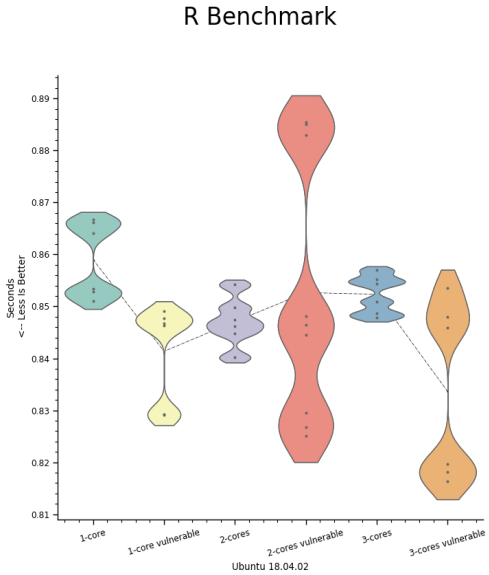

(a) 16 Patches




(b) 18 Patches

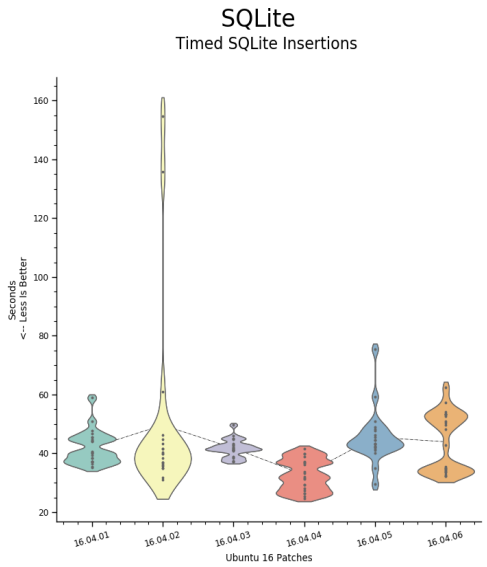



(c) Ubuntu 16.04.06 Cores

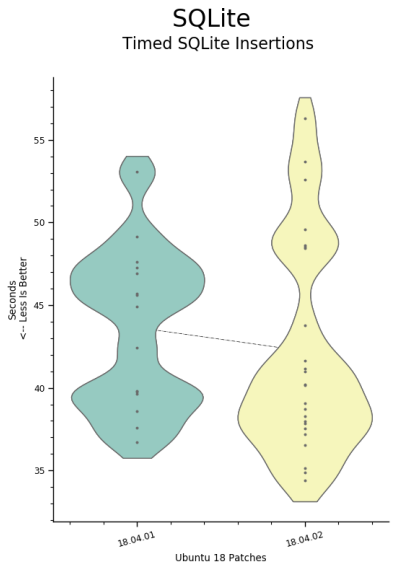



(d) Ubuntu 18.04.2 Cores

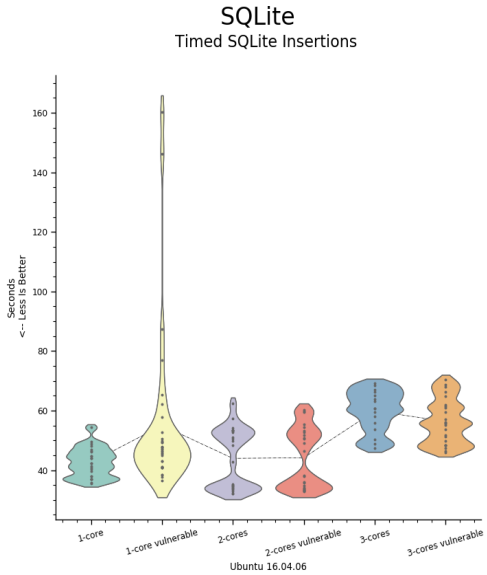
**Figure A.50: R Benchmark Results**



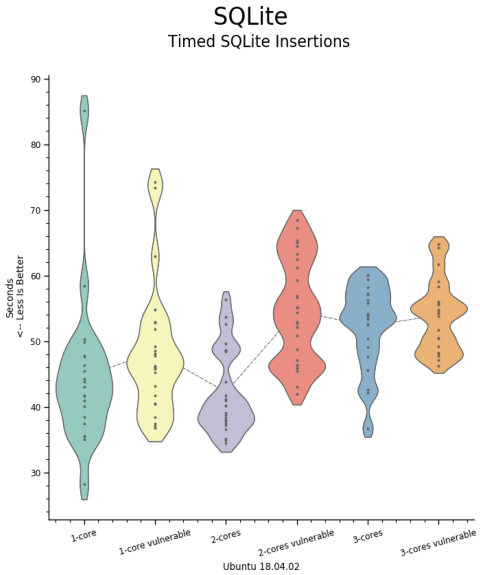

(a) 16 Patches




(b) 18 Patches

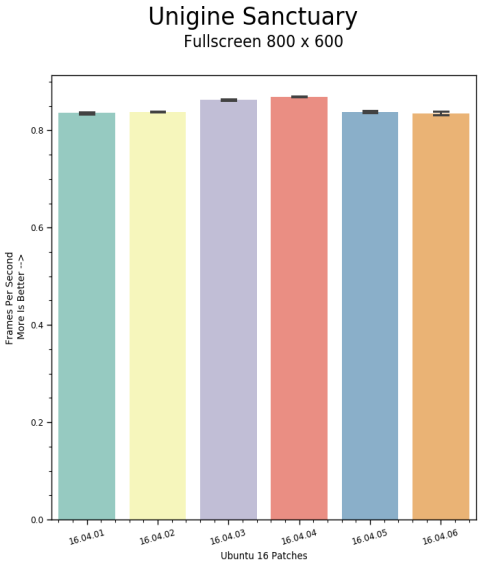



(c) Ubuntu 16.04.06 Cores

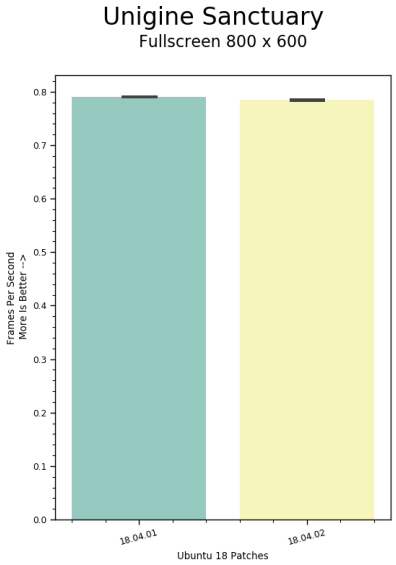



(d) Ubuntu 18.04.2 Cores

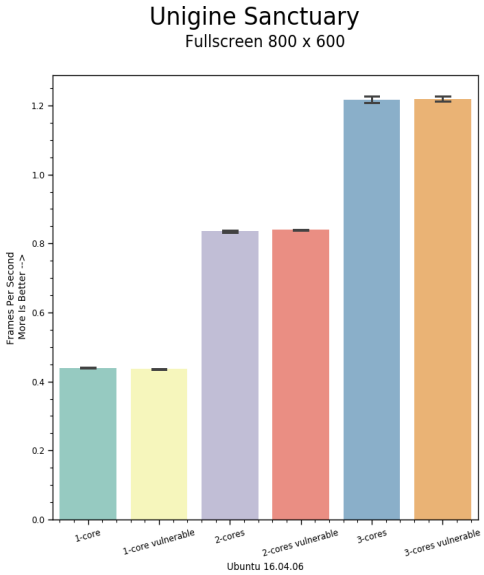
**Figure A.51: SQLite Results**



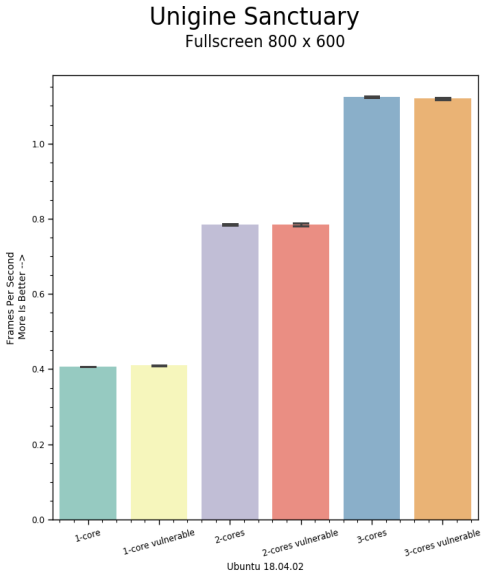

(a) 16 Patches




(b) 18 Patches

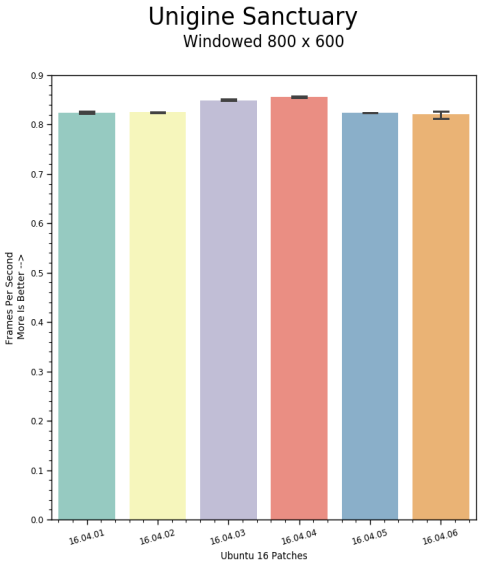



(c) Ubuntu 16.04.06 Cores

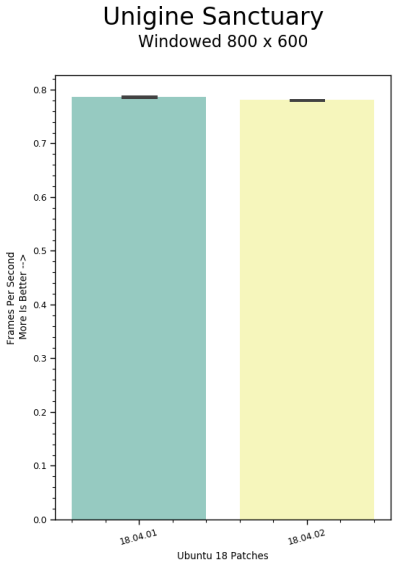



(d) Ubuntu 18.04.2 Cores

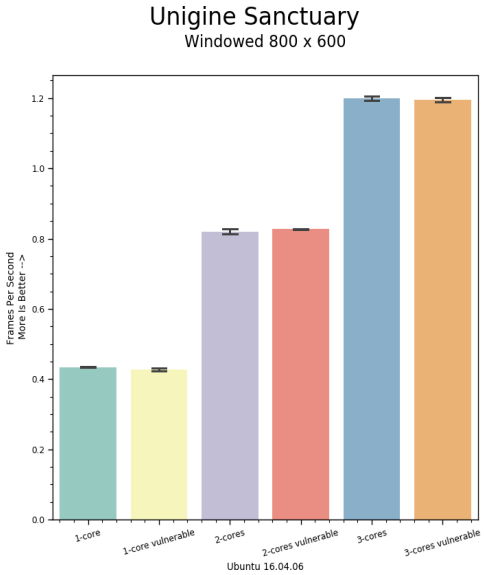
**Figure A.52: Unigine Fullscreen Results**



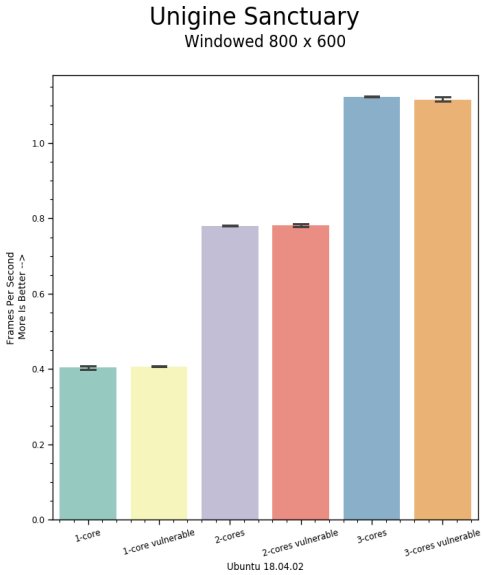

(a) 16 Patches




(b) 18 Patches




(c) Ubuntu 16.04.06 Cores




(d) Ubuntu 18.04.2 Cores

**Figure A.53: Unigine Windowed Results**